

Engintime



Linux

内核实验教程

英真时代 编著

北京英真时代科技有限公司

Linux 内核实验教程

英真时代 编著

版本 1.1

北京英真时代科技有限公司

内容简介

本书结合操作系统原理以及赵炯博士编写的《Linux 内核完全注释》一书，详细分析了一个比较适合在教学中使用的操作系统——Linux 0.11 操作系统的源代码。本书从 Linux 0.11 操作系统中引用了丰富的代码实例，并配以大量的图示，一步一步地引导读者分析 Linux 0.11 操作系统的源代码。本书与其它操作系统理论书籍最明显的不同是，配有若干个精心设计的实验。读者可以亲自动手完成这些实验，在实践的过程中循序渐进地学习 Linux 0.11 操作系统，进而加深对操作系统原理的理解。

本书前 2 章是基础知识，后面配有 12 个实验题目。是一本真正能够引导读者动手实践的书。适合作为高等院校操作系统课程的实践教材，也适合各类程序开发者、爱好者阅读参考。

版权与授权声明

北京英真时代科技有限公司保留本电子书籍的修改和正式出版的所有权利。

您可以从北京英真时代科技有限公司免费获得本电子书籍，但是授予您的权利只限于使用本电子书籍实现个人学习和研究的目的，在未获得北京英真时代科技有限公司许可的情况下，您不能以任何目的传播、复制或抄袭本书之部分或全部内容。

© 2008-2014 北京英真时代科技有限公司，保留所有权利。

北京英真时代科技有限公司联系方式

地址：北京市房山区拱辰大街 98 号 7 层 0825

邮编：102488

电话：010-60357081

QQ：964515564

邮箱：support@tevation.com

网址：<http://www.engintime.com>

目录

第 1 章 Linux Lab概述	4
1.1 Linux 0.11 操作系统.....	4
1.2 集成实验环境.....	4
1.3 从源代码到可运行的操作系统.....	6
1.4 Bochs.....	7
第 2 章 Linux 0.11 编程基础	9
2.1 Linux 0.11 内核源代码的结构.....	9
2.2 NASM汇编.....	10
2.3 C和汇编的相互调用.....	11
2.4 原语操作.....	14
2.5 C语言中变量的内存布局.....	15
2.6 字节顺序Little-endian与Big-endian.....	21
2.7 使用工具阅读Linux 0.11 源代码.....	22
实验一 实验环境的使用.....	25
实验二 操作系统的启动.....	39
实验三 Shell程序设计.....	48
实验四 系统调用.....	55
实验五 进程的创建.....	61
实验六 进程的状态和进程调度.....	68
实验七 信号量的实现和应用.....	81
实验八 地址映射与内存共享.....	88
实验九 页面置换算法与动态内存分配.....	100
实验十 字符显示的控制.....	106
实验十一 proc文件系统的实现.....	114
实验十二 MINIX 1.0 文件系统的实现.....	118
参考文献	123

第 1 章 Linux Lab 概述

本章简要介绍 Linux 0.11 操作系统和与其配套的集成实验环境软件 Linux Lab。阅读本章内容是学习 Linux 0.11 操作系统，并使用 Linux Lab 完成操作系统实验的基础。学习 Linux 0.11 最好的参考资料是赵炯博士编写的《Linux 内核完全注释》，其电子书可以从 Old Linux 网站下载。

1.1 Linux 0.11 操作系统

Linux 0.11 是 Linux 早期可以正常运行的一个内核版本，其内核源代码只有 325KB 左右，其中包含的内容基本上都是 Linux 的精髓。虽然读者在使用 Linux 0.11 的过程中会遇到很多不完善之处，但是并不会影响使用其作为一款教学操作系统，相反，正是由于这些不完善之处，才给了读者更多的想象空间。

Linux 0.11 的源代码主要使用 C 语言编写，有少量的汇编语言代码。Linux 0.11 开放了全部源代码，并配有大量的英文注释。本书用到的 Linux 0.11 又在此基础上又添加了相对应的中文注释，让阅读和理解 Linux 0.11 源代码更加容易。

Linux 0.11 操作系统处于 X86 硬件平台和 Linux 0.11 应用程序之间（如图 1-1 所示），并扮演了极其重要的角色。一方面，Linux 0.11 操作系统对 X86 平台中的各种硬件进行统一的管理，提高了系统资源的利用率。另一方面，Linux 0.11 操作系统提供了一个“虚拟机”和一组系统调用函数，Linux 0.11 应用程序通过这些系统调用函数获得操作系统的服务，从而可以在此“虚拟机”上运行。



图 1-1：Linux 0.11 操作系统处于 X86 硬件平台和 Linux 0.11 应用程序之间

1.2 集成实验环境

Linux Lab 是使用 Linux 0.11 进行操作系统实验的 IDE (Integrated Development Environment)。该 IDE 环境可以直接在 Windows 操作系统上安装和卸载，用户界面和操作习惯与 Microsoft Visual Studio 完全类似，有经验的读者可以迅速上手。正因为有了这样一个易于使用的 IDE 环境，读者可以避免由于手工构建实验环境所带来的学习成本，从而可以将主要精力放在对操作系统原理和 Linux 0.11 源代码的分析与理解上。

使用该 IDE 环境提供的强大功能可以编辑、编译和调试 Linux 0.11 源代码，如图 1-2 所示。编辑功能可以用来阅读和修改 Linux 0.11 源代码；编译功能可以将 Linux 0.11 源代码编译为二进制文件（包括引导程序和内核）；调试功能可以将编译好的二进制文件写入一个软盘镜像（或软盘），然后让虚拟机（或裸机）运行此软盘中的 Linux 0.11，并对其进

远程调试。IDE 环境提供的调试功能十分强大，包括设置断点、单步调试，以及在中断发生时显示对应位置的 C 源代码、查看或修改表达式的值、显示调用堆栈和指令对应的汇编代码等功能。灵活运用各种调试功能对分析 Linux 0.11 的源代码有很大帮助。

IDE 环境除了提供以上的主要功能外，还提供了一些工具软件。使用 Floppy Image Editor 工具提供的可视化用户界面，可以像编辑软盘驱动器中的软盘一样来编辑软盘镜像文件，从而可以在 Windows 操作系统中直接观察到 Linux 0.11 操作系统对软盘的修改。IDE 环境还能与 Bochs 虚拟机软件进行无缝融合，在调试时自动启动 Bochs 虚拟机来运行 Linux 0.11 操作系统。

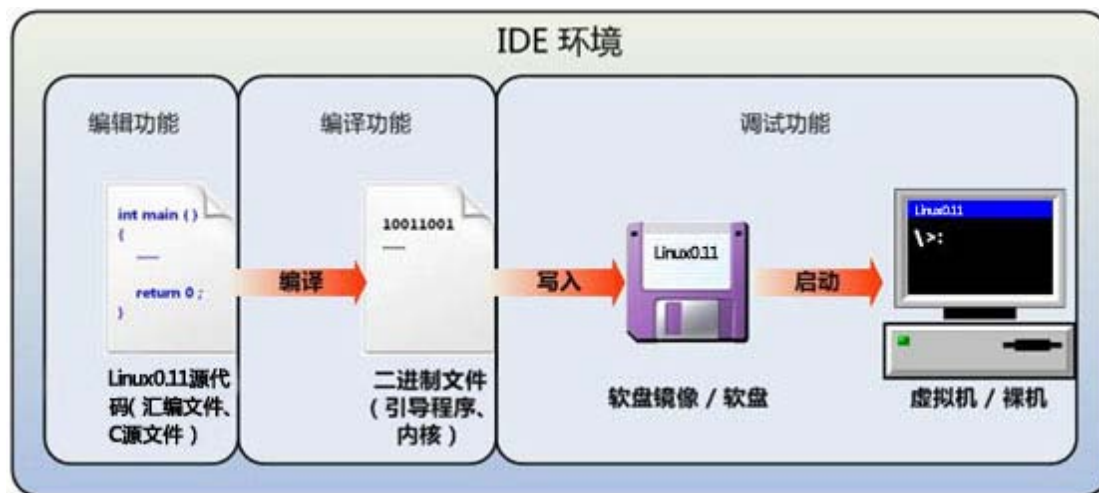


图 1-2: 使用 IDE 环境编辑、编译和调试 Linux 0.11 源代码

Linux 0.11 内核与 IDE 环境组成了“Linux 内核集成实验环境 Linux Lab”（简称 Linux Lab）。图 1-3 显示了读者在进行操作系统实验时，通过使用 IDE 环境编辑、编译、调试 Linux 0.11 源代码，从而在动手实践的过程中达到理解操作系统原理的目的。

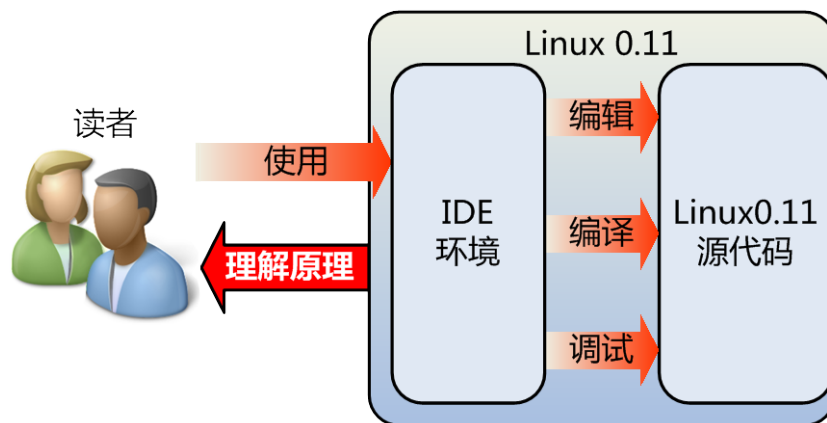


图 1-3: 使用 Linux Lab 进行操作系统实验

Linux Lab 使用的 Linux 0.11 系统与赵炯博士提供的原版的 Linux 0.11 系统有所不同。主要体现在以下几个方面：

- 原版生成的内核文件使用 aout 格式，导致无法使用最新版本的 GDB 调试内核源代码，即便能够使用 Bochs 虚拟机进行调试，但是由于原版生成的调试信息不准确，导致在单步调试源代码时经常发生无法预测的跳跃或异常，严重影响调试过程。Linux Lab 重新优化了所有 C 源文件的编译器选项，并使用 Windows 平台的 PE 格式生成内核文件，这样就生成了能够被最新版本 GDB 识别的完整调试信息，从而允

许在 Windows 平台上使用 GDB 交叉调试内核源代码。并且, 由于生成的调试信息能够准确的将源代码和二进制指令一一对应, 从而允许读者更加准确无误的单步调试源代码, 方便读者通过调试源代码来正确理解操作系统的行为。

- 原版需要使用 vi 工具编写 Linux 0.11 应用程序的源代码文件, 然后使用 GCC 1.4 工具链生成 aout 格式的可执行文件, 操作十分不便。Linux Lab 成功将 GCC 1.4 工具链移植到了 Windows 平台, 从而允许在 Windows 中编写应用程序的源代码文件, 然后交叉编译出可在 Linux 0.11 上运行的 aout 格式的可执行文件, 大大简化读者为 Linux 0.11 开发应用程序的过程。
- 原版使用晦涩难懂的 AT&T 汇编语言编写引导程序(bootsect)和加载程序(setup), 这与国内读者学习的 IBM 汇编语言相差较大。Linux Lab 使用与 IBM 汇编语法更加类似的 NASM 汇编语言对引导程序和加载程序进行了重写, 更加方便读者学习操作系统的引导和加载过程。
- Linux Lab 在原版英文注释的基础上重新编写了大量的中英文对照的注释, 方便读者阅读源代码。
- 原版在终端输出大量字符时经常发生缓冲错误, 导致花屏, 需要频繁清理屏幕, 严重影响使用效果。Linux Lab 修改了源代码中的 BUG, 当在终端输出大量字符时也无需再使用清屏命令。
- 原版中无法正常使用 chmod、mkdir、rm 等文件操作命令, 导致使用者几乎无法完成一般的文件操作。Linux Lab 修改了源代码中的 BUG, 使这些命令都可以正常执行, 从而允许读者可以通过使用这些命令, 来练习 Linux 中的基本文件操作。
- 原版内核提供的 strcmp、strcpy 等函数存在 BUG, 会导致读者在修改内核的过程中产生一些很难定位的错误。Linux Lab 重写了这些函数的源代码, 使读者在修改内核时可以正常调用这些函数。
- 原版使用命令行工具访问存储在软盘 B 中 FAT12 文件系统内的文件, 操作十分不便。Linux Lab 提供的 Floppy image editor 工具使用图形界面编辑软盘 B 中的文件, 操作十分方便。

1.3 从源代码到可运行的操作系统

接下来看看 Linux 0.11 操作系统内核从源代码变为可以在虚拟机上运行的过程, 参见图 1-4。在 Linux Lab 将 Linux 0.11 操作系统内核包含的源代码文件生成为二进制文件的过程中, 会将 bootsect.asm 文件生成为 bootsect.bin 文件(软盘引导扇区程序), 将 setup.asm 文件生成为 setup.bin 文件(加载程序), 将 head.s 和其它源代码文件生成为 linux011.bin 文件。其中 linux011.bin 文件是 Linux 0.11 操作系统的内核。

在 Linux Lab 生成 Linux 0.11 内核项目的最后阶段, 会自动将 bootsect.bin、setup.bin 和 linux011.bin 三个二进制文件写入软盘镜像文件 floppy.a.img 中, 并将此软盘镜像文件

装入虚拟机的软盘驱动器 A 中。启动调试后，虚拟机会运行软盘驱动器 A 中的 Linux 0.11 操作系统。

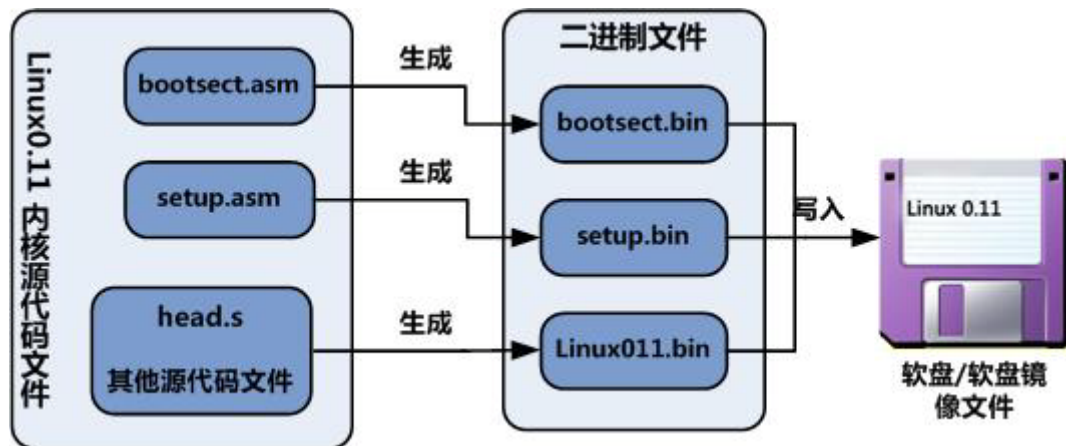


图 1-4: Linux 0.11 操作系统内核从源代码变为可以在虚拟机上运行的过程

1.4 Bochs

Linux 内核集成实验环境 Linux Lab 使用虚拟机工具 Bochs 来运行 Linux 0.11 操作系统，这里对 Bochs 这种虚拟机工具软件进行简单的介绍。

Bochs

Bochs 是一款使用 C++ 语言编写的开源 IA-32 (x86) PC 模拟器，完全使用软件模拟了 Intel x86 CPU、通用 I/O 设备以及可定制的 BIOS 程序。这种完全使用软件模拟的方式又被叫做仿真。所以，准确的说 Bochs 应该是一个仿真器 (Emulator)，而不是虚拟机 (Virtual machine)。大多数操作系统都可以在 Bochs 上运行，例如 Linux, DOS, Windows 95/98/NT/2000/XP/Vista。

由于 Bochs 完全使用软件模拟 X86 硬件平台，所以可以用来调试 BIOS 程序和操作系统的引导程序。也就是说，Bochs 可以从 CPU 加电后执行的第一条指令（也就是 BIOS 程序的第一条指令）处开始调试。Linux Lab 正是利用了 Bochs 的这个特点，使用 Bochs 调试软盘引导扇区程序和加载程序。但是，正是由于 Bochs 完全使用软件来模拟硬件平台，造成其运行时占用大量 CPU 资源且性能较差。

为了使 Bochs 能够更好的调试 Linux 0.11 操作系统，对 Bochs 的源代码进行了必要的修改，重新编译生成了能够与 Linux Lab 无缝融合的 Bochs 版本。所以，必须使用与 Linux Lab 一起提供的 Bochs，而不能直接使用 Bochs 官方提供的安装包。

Bochs 常用的调试命令可以参见下面的表格：

类型	命令	操作
执行控制	c	继续执行，遇到下一个断点后中断。
	s	逐指令调试，执行完下一个指令后中断。可以进入中断服务程序、子程序等。
	n	逐过程调试，执行完下一个过程后中断。
	q	结束调试。
断点	vb segment:offset	在指定的段地址 (segment) 和偏移地址 (offset) 处添加一个断点。

	pb address	在指定的物理地址 (address) 处添加一个断点。
	d n	删除指定序号 (n) 的断点。
	info break	显示当前所有断点的状态信息, 包括各个断点的序号。
寄存器	r	列出 CPU 中的通用寄存器和它们的值。
	creg	列出 CPU 中的控制寄存器和他们的值。
	sreg	列出 CPU 中的段寄存器和他们的值。
内存	x /nuf segment:offset	显示从指定的段地址和偏移地址处开始的内存中的数据, 其中 n 用数字代替, 表示数量, u 可以用 b 代替, 表示以字节为单位, f 指定数据的表示方式, 默认使用十六进制表示。例如命令 x /1024b 0x0000:0x0000 就是显示从段地址 0x0000 偏移地址 0x0000 处开始的 1024 个字节。
	xp /nuf address	显示从指定物理地址处开始的内存中的数据。/nuf 和 x 命令中的用法一致。
	calc register:offset	将参数指定的逻辑地址 (由段寄存器和偏移组成) 转换为线性地址。
	u /n	显示从当前中断位置开始的 n 条指令的反汇编代码。

第 2 章 Linux 0.11 编程基础

本章主要介绍在 Linux 0.11 源代码中涉及到的 C 语言、汇编语言、数据结构等一些基础知识，并在最后简要介绍了使用工具来提高阅读 Linux 0.11 源代码效率的方法。学习本章内容对阅读并理解 Linux 0.11 源代码有很大帮助，建议读者先通读本章内容，尽量掌握本章的知识。如果有一些知识暂时理解不了也没关系，在读者学习后面章节的过程中，可以随时再回到本章学习相关的内容。

2.1 Linux 0.11 内核源代码的结构

使用 Linux Lab 打开 Linux 0.11 内核项目后，在“项目管理器”窗口中可以看到如图 2-1 所示的树结构，在此树结构中可以浏览 Linux 0.11 内核的所有源代码文件。树的根节点表示项目（项目名称为“linux011”），根节点的子节点是项目包含的文件夹或者文件。展开文件夹，可以浏览文件夹所包含的文件。Linux 0.11 内核的源代码文件按照其所属的模块或其实现的功能，组织在不同的文件夹中，详细的说明可以参见表 2-2。

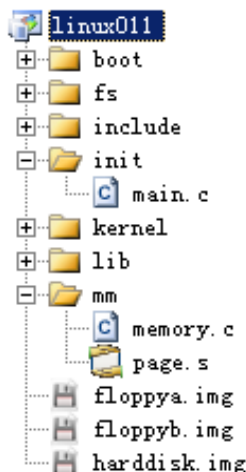


图 2-1: Linux 0.11 内核源代码的树结构

文件夹	说明
boot	引导启动程序目录，包括磁盘引导程序 bootsect.asm、32 位运行启动代码程序 head.s 和获取 BIOS 中参数的 setup.asm 汇编程序
fs	文件系统程序
include	头文件主目录，包括所有的头文件
init	包括内核系统的初始化程序 main.c
kernel	内核程序主目录，包括进程调度、系统调用函数以及 blk_dev 目录下的块设备驱动程序、chr_dev 目录下的字符设备驱动程序和 math 目录下的数学协处理器仿真程序
lib	内核库函数目录，包括向编译系统等提供接口函数的库函数
mm	内存管理程序目录，包括内存管理模块程序
floppya.img	无文件系统的平坦式软盘镜像，大小为 1.44MB，第一扇区为磁盘引导程序 bootsect 模块，第二至第五扇区为获取 BIOS 参数的 setup 模块，其余扇区

	为系统 kernel 模块
floppyb.img	FAT12 文件系统软盘镜像，大小为 1.44MB，用于在 Linux 与 windows 之间交换文件
harddisk.img	MINIX 1.0 文件系统硬盘镜像，提供根文件系统，并存储 Linux 文件

表 2-2: Linux 0.11 内核源代码组织方式的详细说明

在后面的内容中，为了能够向读者准确说明源代码文件的位置，会使用路径格式来表示，例如，init/main.c 就是指在 init 文件夹中的 main.c 文件。

2.2 NASM 汇编

如果读者曾使用 MASM 编写程序，那么这里阐述的 MASM 与 NASM 之间的主要区别可以帮助读者迅速掌握 NASM 的用法。如果读者从来没有学习过汇编语言的相关知识，可以立即跳过本节和下一节，阅读本书并不要求读者必须具备汇编语言方面的知识。

NASM 是大小写敏感的

一个简单的区别是 NASM 是大小写敏感的。当使用符号“foo”，“Foo”或“F00”时，它们是不同的。

NASM 需要方括号来引用内存地址中的内容

先来看看在 MASM 中是怎么做的，比如，如果声明了：

```
foo equ 1
bar dw 2
```

然后有两行代码：

```
mov ax, foo
mov ax, bar
```

尽管它们有看上去是完全相同的语法，但 MASM 却为它们产生了完全不同的操作码。

NASM 为了在看到代码时就能知道会产生什么样的操作码，使用了一个相当简单的内存引用语法。规则是任何对内存中内容的存取操作必须要在地址上加上方括号，但任何对地址值的操作则不需要。所以，形如“mov ax, foo”的指令总是代表一个编译时常数，无论它是一个“EQU”定义的常数或一个变量的地址；如果要取变量“bar”的内容，必须编写代码为“mov ax, word [bar]”。

这就意味着 NASM 不需要 MASM 的“OFFSET”关键字，因为 MASM 的代码“mov ax, offset bar”同 NASM 的“mov ax, word [bar]”是完全等效的。

NASM 同样不支持 MASM 的混合语法。比如 MASM 的“mov ax, table[bx]”语句使用一个中括号外的部分加上一个中括号内的部分来引用一个内存地址，NASM 的语法应该是“mov ax, word [table+bx]”。同样，MASM 中的“mov ax, es:[di]”在 NASM 中应该是“mov ax, word [es:di]”。

NASM 不存储变量的类型

NASM 不会记住声明的变量的类型。然而，MASM 在看到“var dw 0”时会记住类型，也就是声明 var 是一个字大小的变量，然后就可以隐式地使用“mov var, 2”给变量赋值。NASM 不会记住关于变量 var 的任何东西，除了它的起始位置，所以必须显式地编写代码“mov word

[var], 2”。

因此，NASM 不支持“LODS”，“MOVS”，“STOS”，“SCANS”，“CMPS”，“INS”或“OUTS”指令，仅仅支持形如“LODSB”，“MOVSW”和“SCANS”之类的指令。它们都显式地指定了被处理的字符串大小。

NASM 不支持内存模型

NASM 同样不含有任何操作符来支持不同的 16 位内存模型。在使用 NASM 编写 16 位代码时，必须自己跟踪哪些函数需要 FAR CALL，哪些需要 NEAR CALL，并有责任确定放置正确的“RET”指令（“RETN”或“RETF”，NASM 接受“RET”作为“RETN”的另一种形式）；另外必须在调用外部函数时在需要的地方编写 CALL FAR 指令，并必须跟踪哪些外部变量定义是 FAR，哪些是 NEAR。

NASM 不支持 PROC

在 NASM 中使用 PROC 关键字编写函数时，会在函数的开始自动添加代码：

```
push bp
```

```
move bp, sp
```

还会在函数结尾的 ret 指令前自动添加 leave 指令。但是由于 NASM 不支持 PROC 关键字，所以，以上由 NASM 自动添加的代码在 NASM 中必须手动添加。

NASM 可以生成 BIN 文件

NASM 除了可以将 ASM 文件汇编成目标文件（用于与其它目标文件链接成可执行文件）外，还可以将 ASM 文件汇编成一个只包含编写的代码的 BIN 文件。BIN 文件主要用于制作操作系统的引导程序和加载程序，例如由 Linux 0.11 内核源代码文件 boot/bootsect.asm 和 boot/setup.asm 所生成的 boot.bin 和 setup.bin 文件。

用于生成 BIN 文件的 ASM 文件的第一行语句往往会使用 org 关键字，例如“org 0x7C00”。此行语句告诉 NASM 汇编器，这段程序生成的 BIN 文件将要被加载到内存偏移地址 0x7C00 处，这样 NASM 汇编器就可以根据此偏移地址定位程序中变量和标签的位置。

NASM 还经常会用到\$和\$\$。\$表示当前位置被汇编过的地址，所以，语句“jmp \$”表示不停地执行本行指令，也就是死循环。\$\$表示一段程序的开始处被汇编过的地址。在 Linux 0.11 的软盘引导扇区程序源文件（boot/bootsect.asm）中 \$\$ 就表示引导程序的开始地址。所以，在该文件末尾的语句

```
times 510-($-$$) db 0
```

表示将 0 这个字节重复 510-(\$-\$\$) 遍，也就是从当前位置开始，一直到程序的第 510 个字节都填充 0。

另外，NASM 中宏与操作符的工作方式也与 MASM 完全不同，更详细的内容请参考 NASM 汇编器的配套手册。

2.3 C 和汇编的相互调用

操作系统是建立在硬件上的第一层软件，免不了要直接操作硬件，而操作硬件的唯一办法就是使用汇编语言。Linux 0.11 中只包含了极少量的汇编代码，这些汇编代码将一些基本的硬件操作包装成可供 C 语言调用的函数。本节内容主要介绍 C 和汇编在相互调用时应该遵守的一些约定，Linux 0.11 中的代码也同样遵守这些约定。下面示例中的汇编代码使用的是 NASM 汇编语法，如果读者有不明白的地方，可以参考 2.2 节。

在编译 C 代码时，编译器会先将 C 代码翻译成汇编代码，然后再将汇编代码编译成可在硬件上执行的机器语言。下面简单介绍各种 C 编译器在将 C 代码翻译成汇编时所遵守的几项约定：

1. 全局变量的名称和函数名在翻译成汇编符号时，要在名称的前面添加一个下划线。
2. 通过调用栈（Call Stack）传递函数参数。函数参数入栈的顺序是从右到左。
3. 被调用函数（Callee）返回后，由调用者（Caller）释放函数参数占用的栈空间。
4. 通过 EAX 寄存器传递函数的返回值。
5. 被调用函数在使用 EBX、ESI、EDI、EBP 寄存器前要先保存这些寄存器的值，在函数返回时还要恢复这些寄存器的值。被调用函数可以随意使用 EAX、ECX、EDX 寄存器。

C 代码	NASM 汇编代码
<pre>// 定义全局变量 c。 int c = 0; // 定义求和函数。 int add(int a, int b) { // 定义局部变量 int c; c = a + b; return c; } int main() { c = add(5, 6); return 0 }</pre>	<pre>[section .data] ; 数据段 _c dd 0 [section .text] ; 代码段 _add: ; 构造调用栈帧（Call Stack Frame）。 push ebp ; 保存ebp mov ebp, esp ; 新的ebp指向栈顶 sub esp, 4 ; 在栈顶为局部变量c分配空间 L2: mov eax, [ebp + 8] ; eax = a。通过ebp访问参数 mov ecx, [ebp + 12] ; ecx = b add eax, ecx ; 求和 mov [ebp - 4], eax ; 将和赋值给局部变量c mov eax, [ebp - 4] ; 将返回值赋值给eax leave ; 销毁调用栈帧，相当于： ; mov esp, ebp ; pop ebp ret _main: ; 构造调用栈帧。 push ebp mov ebp, esp L1: push dword 6 ; 参数二入栈 push dword 5 ; 参数一入栈 call _add ; 调用函数 L3: add esp, 8 ; 释放参数占用的栈空间 mov [_c], eax ; 将函数返回值赋值给全局变量 c xor eax, eax ; eax = 0 leave ; 销毁调用栈帧 ret }</pre>

表 2-3：C 代码对应的 NASM 汇编代码。

表 2-3 使用一段非常简单的 C 程序和其对应的 NASM 汇编代码来举例说明上面的各项约定。仔细观察表 2-3 中的代码可以发现，C 代码中的全局变量 `c` 和函数名称 `add` 到了汇编代码中都在其前面添加了一个下划线，这是符合第一条调用约定的。在 `main` 函数和 `add` 函数返回时都将返回值放入了 `EAX` 寄存器中，这又符合了调用约定四。调用约定二和约定三规定了函数调用与堆栈协同工作的方式，接下来结合图示进行详细说明。

在开始之前，需要再强调一下关于堆栈的几个知识点：堆栈是一个后进先出的队列，X86 CPU 从硬件层次就支持堆栈操作，提供了 `SS`、`ESP`、`EBP` 等寄存器，其中 `SS` 寄存器保存了堆栈段的起始地址，`ESP` 寄存器保存了栈顶地址。X86 CPU 还提供了用于操作堆栈的指令 `PUSH`、`POP` 等。由于 X86 CPU 的堆栈是从高地址向低地址生长的，所以，`PUSH` 指令会先减少 `ESP` 寄存器的值（CPU 在 16 位实模式下减 2，在 32 位保护模式下减 4），再将数据放入栈顶，`POP` 指令会先从栈顶取出数据，再增加 `ESP` 寄存器的值。强调了关于堆栈的基本知识，再结合示例程序在 32 位保护模式下执行的过程，详细讨论函数调用对堆栈的影响。

当程序执行到 `main` 函数的 L1 行代码时，可以认为这是调用堆栈的初始状态，此时 `EBP` 和 `ESP` 同时指向栈顶，如图 2-4。

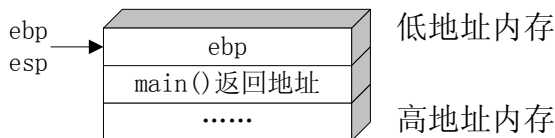


图 2-4：初始的堆栈状态

在调用 `add` 函数前，首先将函数的参数按照从右到左的顺序压入堆栈，如图 2-5。

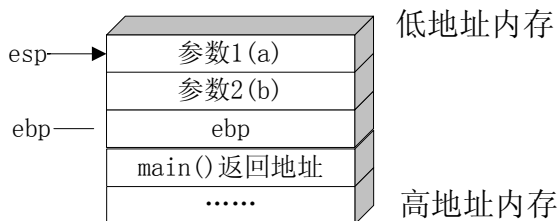


图 2-5：参数入栈后的堆栈状态

使用 `CALL` 指令调用 `add` 函数，会首先将 `add` 函数的返回地址压入堆栈，然后再跳转到 `add` 函数的起始地址继续执行，所以当执行到 `add` 函数的 L2 行代码时，堆栈状态如图 2-6。

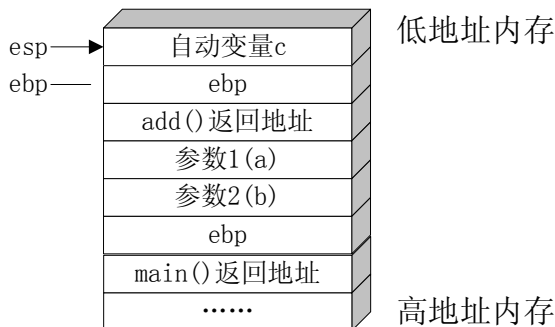


图 2-6：进入函数后的堆栈状态

由于在参数入栈后，又先后有 `add` 函数的返回地址和 `EBP` 寄存器入栈，所以 `EBP+8` 是参数 1

的起始地址，EBP+12 是参数 2 的起始地址。在 add 函数结束时，LEAVE 指令会将 EBP 赋值给 ESP，然后再将 EBP 出栈，也就是恢复旧的 EBP 中的值，此时堆栈状态如图 2-7。

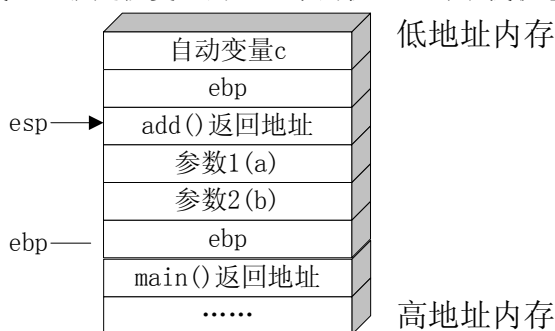


图 2-7：LEAVE 指令执行后的堆栈状态

RET 指令会将 add 函数的返回地址出栈并放入 IP 寄存器继续执行，也就是从 main 函数的 L3 行代码处继续执行，此时堆栈的状态如图 2-8。

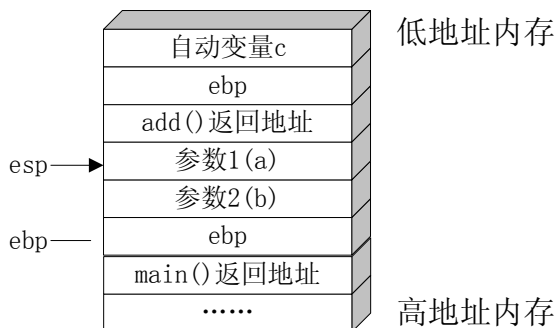


图 2-8：RET 指令执行后的堆栈状态

最后，为了清理堆栈中传递给 add 函数的参数，还需要在 main 函数中为 ESP 增加 8，从而回到图 2-8 所示的状态。至此，调用约定二和调用约定三也讲解完毕。

弄明白了调用约定，在 C 中调用汇编函数就很简单了。先按照调用约定编写汇编函数，然后在汇编文件中使用 global 关键字将汇编函数符号声明为全局的，最后在 C 源文件中声明汇编函数对应的 C 语言函数原型，即可在 C 中像调用 C 函数一样调用汇编函数了。

在汇编中调用 C 函数同样简单。完成 C 函数后，在汇编文件中使用 extern 关键字声明 C 函数名称对应的汇编符号，然后，在汇编中按照约定调用即可。

2.4 原语操作

Linux 0.11 内核中维护了大量的内核数据，正是这些内核数据描述了 Linux 0.11 操作系统的状态。如果有一组相互关联的内核数据共同描述了操作系统的某个状态，那么在修改这样一组内核数据时就必须保证它们的一致性，即要么不修改，要么就全都修改。这就要求修改这部分内核数据的代码在执行的过程中不能被打断，这种不能被打断的操作就被称为“原语操作”。

如何保证代码的执行不被打断呢？或者说如何保证一个操作是原语操作呢？这要从软、硬两个方面来解决。首先从软的方面考虑，需要保证编写的代码在修改内核数据的过程中不会中断执行，例如不能在只修改了一部分数据后就结束操作，这只需要在设计原语操作和编写代码的时候多加小心即可。接下来从硬的方面考虑，外部设备发送给 CPU 的中断会让 CPU 暂时中止当前程序的执行，转而执行相应的中断处理程序。现在假设一个原语操作要修改内

核中的日期和时间两个变量，但是在原语操作只修改了日期后就发生了外部中断，中断处理程序从内核中读取的日期和时间肯定就是错误的。所以，一般情况下，在执行原语操作前需要通知 CPU 停止响应外部中断，待操作执行完毕后再通知 CPU 恢复响应外部中断。

X86 CPU 是根据 `eflags` 状态字寄存器中的 `IF` 位来决定是否响应外部中断的，并提供了 `STI` 指令设置 `IF` 位从而允许响应外部中断，还提供了 `CLI` 指令清空 `IF` 位从而停止响应外部中断。

在 Linux 0.11 内核中应该成对使用指令 `STI` 和 `CLI` 来实现原语操作。例如，内核中的函数 `A` 需要实现一个原语操作，就应该按照下面的方式编写代码：

```
void A ()
{
    BOOL IntState; // 定义一个局部变量，用于保存停止中断响应前的中断状态。
    ... // 非原语操作代码。
    CLI; // 停止响应外部中断。
    ... // 原语操作代码。
    STI; // 恢复停止响应外部中断前的中断状态。
    ... // 非原语操作代码。
}
```

Linux 0.11 中不支持原语操作的嵌套。所以下面的代码是有问题的。

```
void B ()
{
    BOOL IntState;
    CLI;
    A(); // 调用函数 A。B 的原语操作包含了 A 的操作内容。
    ...// 其它原语操作
    STI;
}
```

2.5 C 语言中变量的内存布局

C 语言编写的源代码用于描述程序中的指令和数据。其中，指令全部保存在程序的可执行文件中，并随可执行文件一同载入内存。绝大多数情况下，指令在内存中的位置是不变的，并且是只读的，所以暂时不做过多的讨论。程序中的数据主要是由 C 源代码中的各种数据类

型定义的变量来描述的，而且这些数据在内存中的分布情况要复杂一些，这也就是本节要讨论的重点。通过阅读本节内容，读者可以了解到 C 语言定义的数据类型所描述的内存布局，还能够了解到各种变量在内存中的位置，这对于读者深刻理解 Linux 0.11 操作系统的行为，特别是内存使用情况会有很大帮助。

从 CPU 的角度观察，内存就是一个由若干字节组成的一维数组，用来访问数组元素的下标就是内存的地址。与典型数组不同的是，CPU 可以将从某个地址开始的几个字节做为一个整体来同时访问，例如，CPU 可以同时访问 1、2、4 或更多个字节。可以这样来理解，CPU 在访问内存时需要同时具备两个要素：一个是**内存基址**，即从哪里开始访问内存；另一个是**内存布局**，即访问的字节数量。相对应的，在 C 语言编写的源代码中，数据类型（包括基本数据类型和结构体等）用来描述内存布局，并不占用实际的内存，而只有使用这些数据类型定义的变量才会占用实际的内存，从而确定内存基址。

数据类型描述的内存布局

在 C 语言中预定义的基本数据类型，包括 char、short、long 和指针类型等，所描述的内存布局就是若干个连续的字节。例如 char 类型描述了 1 个字节，short 类型描述了 2 个字节，long 类型描述了 4 个字节，指针类型（无论是哪种指针类型）也是描述了 4 个字节。使用这些数据类型定义变量的过程，就是为变量分配内存的过程，也就是确定基址的过程。再结合这些数据类型所描述的内存布局，CPU 即可访问变量所在的内存。考虑下面的代码：

```
char a = 'A';
short b = 0x1234;
long c = 0x12345678;
void* p = &c;
```

这四个变量的内存布局可以像图 2-9 所示的样子（注意字节是反序的，原因参见第 2.6 节）。以变量 p 为例，其内存基址为 0x402008，并且由于空指针类型描述的内存布局是 4 个字节，所以变量 p 所在的内存为从 0x402008 起始的 4 个字节。由于变量 p 所在的内存同时具备了内存基址和内存布局这两个要素，CPU 就可以访问变量 p 所在的内存了，于是 CPU 可以将变量 c 的地址放入变量 p 所在的内存。

接下来分析一下 CPU 在访问内存时，如果缺少了某个要素，会出现什么样的情况。如果添加了一行语句“*p = b;”，则编译器会报告错误。原因是该语句的本意是将变量 b 内存中的数据复制到指针 p 所指向的内存中。虽然已经知道指针 p 指向的内存基址是 0x402004，但是由于指针 p 是一个空指针类型，即 p 所指向的内存的类型为空（void）。所以，CPU 在试图访问指针 p 所指向的内存时，就无法确定从基址开始访问的字节数量，也就是缺少了内存布局这个要素。对于这种情况，可以将语句修改为“*(short*)p = b;”，将指针 p 的类型强制转换为 short 指针类型，即使用 short 类型描述指针 p 所指向的内存，则该语句就可以将 0x402004 字节的值修改为 0x34，将 0x402005 字节的值修改为 0x12 了。于是可以得出一个结论：类型转换（包括自动转换和强制转换）的过程，就是修改内存布局这个要素的过程。

本质上，只要具备了内存基址和内存布局这两个要素，即使不使用变量也同样可以访问内存，例如语句“*(short*)0x402006 = b;”也是可以正确执行的。读者应该学会从内存基址和内存布局的角度来理解各种数据类型（特别是指针类型）的使用方法。当源代码中出现问题或者是难于理解的地方，可以尝试使用内存基址和内存布局这两个要素来进行分析。

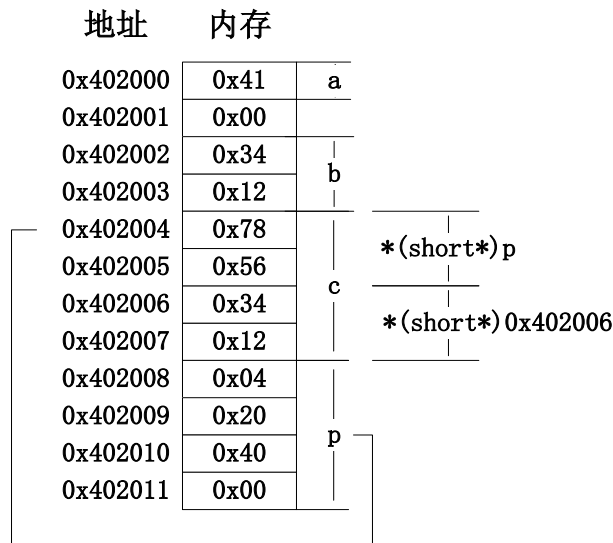


图 2-9：最简单的数据类型所描述的内存布局。

结构体类型定义的变量也同样具有内存基址和内存布局这两个要素，只不过由结构体类型定义的变量，其内存布局还需要遵守以下的两条准则：

- 结构体变量中第一个域的内存基址等于整个结构体变量的内存基址。
- 结构体变量中各个域的内存基址是随它们的声明顺序依次递增的。

接下来通过一些实际的例子来说明这两条准则。为了强调内存基址和内存布局这两个要素，举例时会使用结构体类型的指针指向一块内存，从而确定内存基址和内存布局。首先考虑下面的源代码：

```
unsigned char ByteArray[8] = {0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80};
```

```
typedef struct _F00 {
    long Member1;
    long Member2;
} F00, *PF00;
```

```
PF00 Pointer = (PF00)ByteArray;
```

指针 `Pointer` 指向的内存基址和内存布局可以像图 2-10 所示的样子。此时，表达式 `Pointer->Member1` 的值为 `0x40302010`，表达式 `Pointer->Member2` 的值为 `0x80706050`。其中，表达式 `&Pointer->Member1` 得到的地址为 `0x402000`，与指针 `Pointer` 指向的地址相同，可以说明第一条准则是成立的；表达式 `&Pointer->Member2` 得到的地址大于 `&Pointer->Member1` 得到的地址，可以说明第二条准则也是成立的。

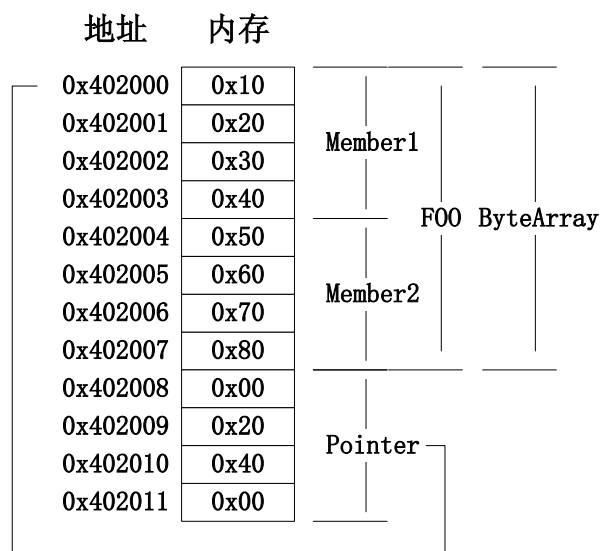


图 2-10：结构体类型所描述的内存布局。

这里需要特别说明一下二元操作符“->”，注意此操作符的左侧和右侧都会涉及到内存基址和内存布局这两个要素。该操作符的工作过程是这样的，首先根据左侧的结构体类型和右侧的域，计算出域在结构体类型中的偏移值，然后将该偏移值与左侧的结构体指针变量所指向的地址相加，从而得到右侧域的内存基址，最后再结合域的数据类型（内存布局）来访问对应的内存。例如，考虑表达式 `Pointer->Member2`，“->”操作符首先计算出域 `Member2` 在结构体 `F00` 中的偏移值是 4，与 `Pointer` 指向的地址 `0x40200` 相加得到地址 `0x40204`，再结合域 `Member2` 的数据类型（内存布局）访问内存中的数据。

结构体中相邻的域所描述的内存布局总是紧密相邻的吗？答案是否定的。为了提升 CPU 访问内存的速度，默认情况下，C 语言编译器会保证基本数据类型的内存基址是某个数 `k`（通常为 2 或 4）的倍数，这就是所谓的**内存对齐**，而这个 `k` 则被称为该数据类型的对齐模数。以用来编译 Linux 0.11 源代码的 GCC 编译器为例，默认情况下，任何基本数据类型的对齐模数就是该数据类型的大小。比如对于 `double` 类型（大小为 8 字节），就要求该数据类型的内存基址总是 8 的倍数，而 `char` 数据类型（大小为 1 字节）的内存基址则可以从任何一个地址开始。考虑下面的两个结构体：

```
// #pragma pack(1)

typedef struct _F001 {
    short Member1;
    long Member2;
} F001, *PF001;

typedef struct _F002 {
    unsigned char Member1;
    short Member2;
    long Member3;
} F002, *PF002;
```

```
// #pragma pack()
```

如果使用这两个结构体定义的指针变量指向 ByteArray 数组,则指针变量描述的内存布局可以像图 2-11 所示的样子。

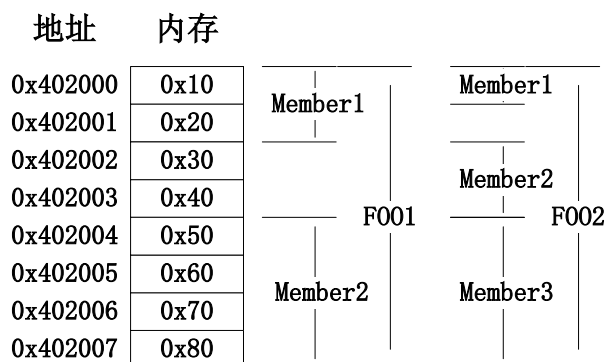


图 2-11: 默认情况下结构体的内存布局。

C 语言提供了一个编译器指令 “#pragma pack(n)” 用来指定数据类型的对齐模数。将 n 替换为指定的对齐模数,则在该编译器指令之后出现的所有数据类型都会使用指定的模数来进行内存对齐,如果忽略了小括号中的 n,就会使用默认的方式来进行内存对齐。所以,如果取消注释之前代码中的第一行和最后一行语句,内存布局就会变为图 2-12 所示的样子。

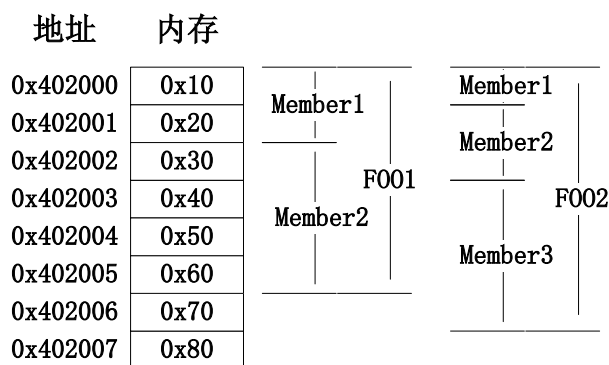


图 2-12: 按照 1 字节对齐后的结构体的内存布局。

接下来说明一下联合体类型所描述的内存布局。联合体类型中的各个域总是使用相同的内存基址,但是它们会使用各自的数据类型来描述内存布局,并且联合体类型的大小由占用字节最多的那个域来决定。下面的代码在结构体类型中嵌入了一个联合体:

```
typedef struct _F003 {
    union {
        short Member1;
        long Member2;
    }u;
    long Member3;
} F003, *PF003;
```

如果使用此结构体定义的指针变量指向 ByteArray 数组,则指针变量描述的内存布局可以像图 2-13 所示的样子。



图 2-13：联合体的内存布局。

最后，由于 Linux 0.11 源代码中还用到了位域这种数据类型，所以再简单介绍一下位域的内存布局（关于位域的详细用法，请读者参考 C 语言程序设计教材）。所谓位域，就是把若干字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。定义位域时，其各个域的数据类型必须是相同的，并且由此数据类型决定整个位域的内存布局（占用的字节数量），而各个域只说明各自占用的位数。考虑下面定义的位域：

```
typedef struct _F004 {
    long Head:10;
    long Middle:10;
    long Tail:12;
} F004, *PF004;
```

如果使用此位域定义的指针变量指向 ByteArray 数组，则指针变量描述的内存布局可以像图 2-14 所示的样子。此位域中各个域都是 long 类型的，所以整个位域可以描述从 0x402000 开始的 4 个字节。此时，表达式 `Pointer->Head` 的值为 0x10 (0000010000)，表达式 `Pointer->Middle` 的值为 0x08 (0000001000)，表达式 `Pointer->Tail` 的值为 0x403 (010000000011)。

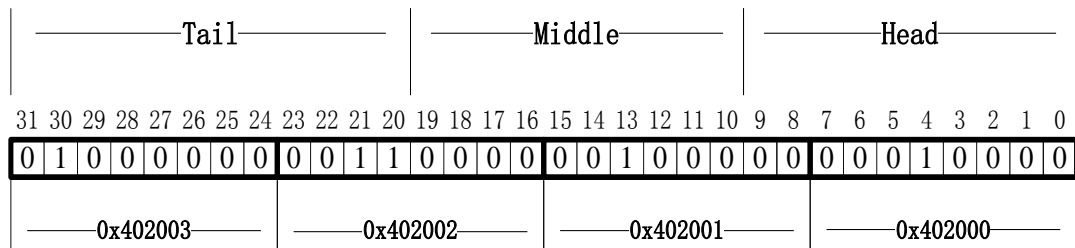


图 2-14：位域的内存布局。

变量在内存中的位置

以变量在内存中的位置来区分，可以将变量分为**静态变量**和**动态变量**。静态变量是指在程序运行期间在内存中的位置不会发生改变的那些变量，包括全局变量和使用 `static` 声明的局部变量。动态变量是指在程序运行期间会动态的为其分配内存的那些变量，包括函数的形式参数和局部变量（未加 `static` 声明）。

相对应的，程序在运行期间所占用的内存也会分为静态存储区和动态存储区。其中，静态存储区与程序可执行文件中的数据区是完全相同的，原因是在使用 C 源代码生成程序的可执行文件时，就已经把所有的静态变量放置在数据区中。在一个程序开始执行时，操作系统会首先将程序的可执行文件载入内存，并使用可执行文件中的数据区创建静态存储区，这样

所有的静态变量就很轻松的出现在内存中了。对于动态存储区，它是在程序开始执行之前被操作系统创建的一块内存，用来存放动态变量以及函数调用时的现场和返回地址，也就是常说的栈（stack）。在程序运行期间，调用函数时会为动态变量分配栈，函数结束时会展开栈（可以参考第 2.3 节）。

2.6 字节顺序 Little-endian 与 Big-endian

字节顺序

这里的“字节顺序”是指，当存放多字节数据（例如 4 个字节的长整型）时，数据中多个字节的存放顺序。典型的情况是整数在内存或文件中的存放方式和网络传输的传输顺序。

对于单一的字节，大部分处理器以相同的顺序处理位元，因此单字节的存放方法和传输方式一般相同。对于多字节数据，在不同的处理器的存放方式主要有两种，一种是 Little-endian，另一种是 Big-endian。

Little-endian

采用此种字节顺序存储长整型数据 0x0A0B0C0D 到内存或文件中是下面的样子：

低地址	
a	0x0D
a+1	0x0C
a+2	0x0B
a+3	0x0A
高地址	

可以简单的记忆为“低字节存在低地址”。

Big-endian

采用此种字节顺序存储长整型数据 0x0A0B0C0D 到内存或文件中是下面的样子：

低地址	
a	0x0A
a+1	0x0B
a+2	0x0C
a+3	0x0D
高地址	

可以简单的记忆为“高字节存在低地址”。

何处使用

网络传输一般使用 Big-endian。而不同的处理器体系会使用不同的字节顺序：

- X86, MOS Technology 6502, Z80, VAX, PDP-11 等处理器为 Little endian。
- Motorola 6800, Motorola 68000, PowerPC 970, System/370, SPARC（除 V9 外）等处理器为 Big endian。
- ARM, PowerPC（除 PowerPC 970 外）, DEC Alpha, SPARC V9, MIPS, PA-RISC and IA64 的字节序是可配置的。

掌握的意义

在多数情况下，不需要关心字节顺序就可以编写程序，但是在某些场合必须知道平台的

字节顺序才能完成工作，例如在采用 Intel X86 处理器的计算机上调试程序，如果想查看一块内存中的数据，就必须意识到数据是使用 Little-endian 的字节顺序存储到内存中的，这样在人工读取的时候才能够识别出正确的数据。

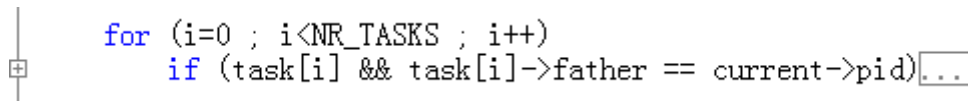
2.7 使用工具阅读 Linux 0.11 源代码

“工欲善其事，必先利其器”。Linux Lab 提供了一些工具，用于帮助读者提高阅读 Linux 0.11 源代码的效率，达到事半功倍的目的。

源代码编辑器

Linux Lab 提供的源代码编辑器为多种源代码文件（.c, .cpp, .h, .asm 等）提供了符号高亮显示功能，可以帮助读者在阅读源代码的过程中，轻松分辨出各种符号（包括关键字、字符串、寄存器、注释等）。在编辑器中还显示了源代码所在的行号，方便读者准确定位源代码的位置。

编辑器为 C 语言代码提供了大纲功能。使用大纲功能，读者可以将一些嵌套较深的源代码折叠起来，帮助读者理解源代码的结构。在图 2-15 中显示了将一个双重循环的第二层循环折叠后的效果。读者在阅读一些大型的代码文件时，如果使用大纲功能将一些不关心的函数或者大段的注释折叠起来，可以使代码看起来更加短小。



```

for (i=0 ; i<NR_TASKS ; i++)
    if (task[i] && task[i]->father == current->pid) ...

```

图 2-15：使用大纲将源代码折叠。

文本查找工具

Linux Lab 提供了一些快捷键，用于在打开的源代码文件中迅速查找文本。例如在一个函数中遇到了一个局部变量 Var，如果想知道定义此变量的数据类型，可以使用鼠标选中此变量的名称，然后按 Ctrl+Shift+F3 快捷键在代码中向上查找相同的文本，如果找到的仍然是使用此变量的代码而不是定义此变量的代码，可以按 Ctrl+Shift+F3 继续向上查找，直到找到定义此变量的代码为止。如果在一个函数的开始遇到了一个局部变量的定义 LONG Var，想知道在此函数中都有哪些地方用到了此局部变量，可以使用鼠标选中此变量的名称，然后重复按 Ctrl+F3 快捷键在代码中向下查找相同的文本，直到浏览了所有用到此变量的代码为止。

Linux Lab 提供了“查找和替换”对话框，用于在多个文件中查找文本。如果在阅读代码时遇到了一个函数调用函数 Fun，想查看此函数是如何实现的（即函数的定义），可以使用鼠标选中函数的名称，然后按 Ctrl+Shift+F 快捷键弹出“查找和替换”对话框，在对话框的“查找内容”中会自动填入选中的函数名称（如图 2-16）。点击“查找全部”按钮后，Linux Lab 会自动激活“查找结果”窗口，并会在其中显示出各个文件中出现此函数名称的位置。在“查找结果”窗口中双击要查看的位置，源代码编辑器会打开源代码文件，并将光标设置在文本所在行。这种方法还可以用来查找所有调用了 Fun 函数的代码行，帮助读者理解该函数的使用方法。

这里需要注意的是，如果读者要查找的函数是在汇编代码中定义的，在 C 语言代码中被使用（或者相反），则函数的名称会不同（参见第 2.3 节），此时需要将“查找和替换”对话框中的“全字匹配”复选框取消选中。

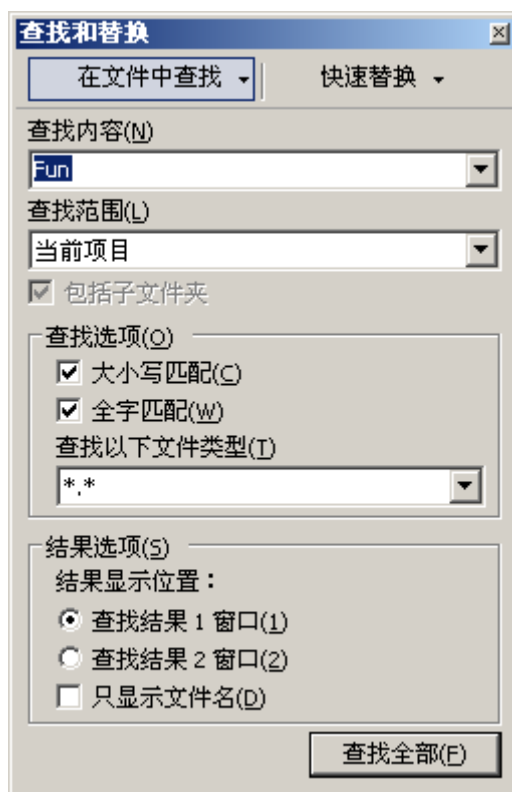


图 2-16: “查找和替换”对话框。

书签工具

书签能够记录读者所关注的代码所在的位置，让读者迅速准确的找到想要查看的代码。假如读者正在研究某个函数的用法，可能需要在调用此函数的代码行和定义此函数的代码行间来回切换，此时读者可以在调用此函数的代码行添加一个书签（左键点击此代码行，然后按 Ctrl+F2 快捷键），然后在定义此函数的代码行添加另一个书签，接下来就可以通过反复按 F2 键在这两段代码间快速切换了。如果读者不再需要这些书签，可以按 Ctrl+Shift+F2 快捷键删除所有书签。更多的书签功能可以在 Linux Lab 中查看“编辑”菜单中的“书签”子菜单。

编译器工具

编译器工具可以帮助读者定位代码中的警告和错误，读者可以适时的使用快捷键 Ctrl+Alt+F7 执行“重新生成项目”来检查代码。如果在 Linux Lab 的“输出”窗口中输出了警告或者错误，可以在“输出”窗口中双击要查看的行，源代码编辑器会打开源代码文件，并将光标设置在警告或错误所在行。

如果生成项目时报告的错误非常多，可以尝试着修改最前面的一两个错误，然后再次重新生成项目，报告的错误往往会减少很多。例如修改了头文件中的语法错误，则所有包含了此头文件的源文件都不会再报告此错误。

调试器工具

要探究程序动态运行时的每个细节，需要在调试器中运行它。虽然调试器主要用于查找程序的错误，它还是分析程序运行的万能工具。下面的列表概括了对阅读代码最有帮助的调试器特性。

- 单步执行允许读者针对给定的输入，跟踪程序执行的精确顺序。调试器允许读者跳过子例程调用（当对特定的例程不感兴趣时可以按 F10 跳过）或者进入调用（当希望分析例程的操作时按 F11 进入）。
- 断点能够在程序执行到特定的点时，让程序停下来。读者可以使用断点快速地跳转

到感兴趣的点，或检查某块代码是否得到执行。

- 数据提示可以为读者提供相应的视图，显示变量的值。使用它们可以监控这些变量在程序运行过程中如何变更。同时能够展开结构的成员，帮助读者对数据结构进行分析、理解和检验。
- 调用堆栈为读者提供通向当前执行点的调用历史，以及每个例程的地址及参数（在 C 和 C++ 中从 main 开始），可以帮助读者理解函数调用的层次。
- 反汇编可以让读者查看例程的汇编代码，可以用于检查 C 语言编写的与硬件相关的代码，是否执行了预期的操作，也可以让读者调试那些没有调试信息的汇编模块。

以上提到的各种工具不单是 Linux Lab 会提供，很多集成开发环境都会提供类似的功能。读者在阅读 Linux 0.11 源代码的过程中应该有意识的多使用这些工具，熟练使用后才能发挥真正的威力。

实验一 实验环境的使用

实验性质：验证

建议学时：2 学时

实验难度：★★☆☆☆

一、 实验目的

- 熟悉 Linux 内核集成实验环境 Linux Lab 的基本使用方法。
- 练习编译、调试 Linux 内核及应用程序。

二、 预备知识

请读者认真阅读本书第 1 章的内容，同时可以阅读赵炯博士编写的《Linux 内核完全注释》一书第 1 章的内容，从而对 Linux 0.11 内核以及 Linux 内核集成实验环境 Linux Lab 有一个初步的认识。

三、 实验内容

3.1 启动 Linux Lab

1. 在安装有 Linux Lab 的计算机上，可以使用两种不同的方法来启动 Linux Lab：
 - 在桌面上双击“Engintime Linux Lab”图标。
 - 或者
 - 点击“开始”菜单，在“程序”中的“Engintime Linux Lab”中选择“Engintime Linux Lab”。
2. Linux Lab 每次启动后都会首先弹出一个用于注册用户信息的对话框。在此对话框中填入学号和姓名后，点击“确定”按钮完成本次注册。
3. 观察 Linux Lab 主窗口的布局。Linux Lab 主要由下面的若干元素组成：菜单栏、工具栏以及停靠在左侧和底部的各种工具窗口，余下的区域用来放置编辑器窗口。

3.2 学习 Linux Lab 的基本使用方法

练习使用 Linux Lab 编写一个 Windows 控制台应用程序，熟悉 Linux Lab 的基本使用方法（主要包括新建项目、生成项目、调试项目等）。

新建 Windows 控制台应用程序项目

新建一个 Windows 控制台应用程序项目的步骤如下：

1. 在“文件”菜单中选择“新建”，然后单击“项目”。
2. 在“新建项目”对话框中，选择项目模板“控制台应用程序（c）”。
3. 在“名称”中输入新项目使用的文件夹名称“lab1”。
4. 在“位置”中输入新项目保存在磁盘上的位置“C:\test”。
5. 点击“确定”按钮。

新建完毕后，Linux Lab 会自动打开这个新建的项目，该项目是一个用 C 语言编写的 Windows 控制台应用程序。在 Linux Lab 的“项目管理器”窗口中（如图 1-1 所示），树的根节点是项目节点，项目的名称是“console”，各个子节点是项目包含的文件夹或者文件。

此项目的源代码主要包含一个头文件“console.h”和一个C语言源文件“console.c”。

可以使用“Windows 资源管理器”打开磁盘上的“C:\test\lab1”文件夹，查看项目中包含的文件（提示，在“项目管理器”窗口的项目节点上点击右键，然后在弹出的快捷菜单中选择“打开所在的文件夹”即可）。

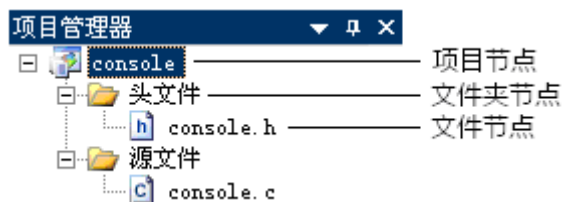


图 1-1：打开 Windows 控制台应用程序项目后的“项目管理器”窗口

生成项目

使用“生成项目”功能可以将程序的源代码文件编译为可执行的二进制文件，方法十分简单：在“生成”菜单中选择“生成项目”。

在项目生成过程中，“输出”窗口会实时显示生成的进度和结果。如果源代码中不包含语法错误，会在最后提示生成成功，如图 1-2：

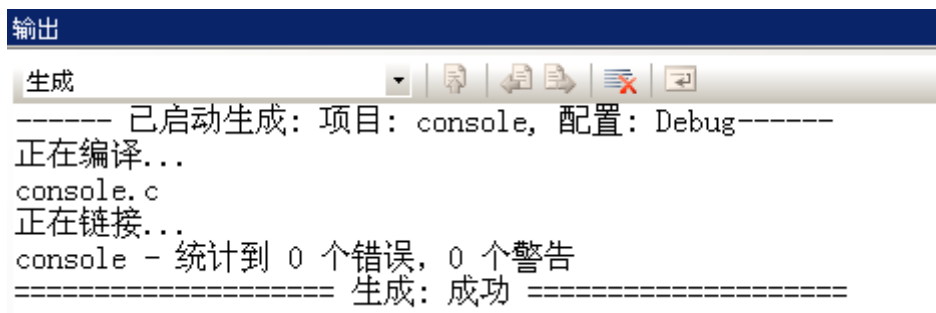


图 1-2：成功生成 Windows 控制台应用程序项目后的“输出”窗口

如果源代码中存在语法错误，“输出”窗口会输出相应的错误信息（包括错误所在文件的路径，错误在文件中的位置，以及错误原因），并在最后提示生成失败。此时在“输出”窗口中双击错误信息所在的行，Linux Lab 会使用源代码编辑器打开错误所在的文件，并自动定位到错误对应的代码行。可以在源代码文件中故意输入一些错误的代码（例如删除一个代码行结尾的分号），然后再次生成项目，然后在“输出”窗口中双击错误信息来定位存在错误的代码行，将代码修改正确后再生成项目。

生成过程是将每个源代码文件（.c、.cpp、.asm 等文件）编译为一个对象文件（.o 文件），然后再将多个对象文件链接为一个目标文件（.exe、.dll 等文件）。成功生成 Windows 控制台应用程序项目后，默认会在“C:\test\lab1\debug”目录下生成一个名称为“console.o”的对象文件和名称为“console.exe”的 Windows 控制台应用程序，可以使用 Windows 资源管理器查看这些文件。

执行项目

在 Linux Lab 中选择“调试”菜单中的“开始执行(不调试)”，可以执行刚刚生成的 Windows 控制台应用程序。启动执行后会弹出一个 Windows 控制台窗口，显示控制台应用程序输出的内容。按任意键即可关闭此 Windows 控制台窗口。

调试项目

Linux Lab 提供的调试器是一个功能强大的工具，使用此调试器可以观察程序的运行时行为并确定逻辑错误的位置，可以中断（或挂起）程序的执行以检查代码，计算和编辑程序中的变量，查看寄存器，以及查看从源代码创建的指令。为了顺利进行后续的各项实验，应该学会灵活使用这些调试功能。

在开始练习各种调试功能之前，首先需要对刚刚创建的例子程序进行必要的修改，步骤如下：

1. 右键点击“项目管理器”窗口中的“源文件”文件夹节点，在弹出的快捷菜单中选择“添加”中的“添加新文件”。
2. 在弹出的“添加新文件”对话框中选择“C 源文件”模板。
3. 在“名称”中输入文件名称“func”。
4. 点击“添加”按钮，添加并自动打开文件 func.c，此时的“项目管理器”窗口会如图 1-3 所示：

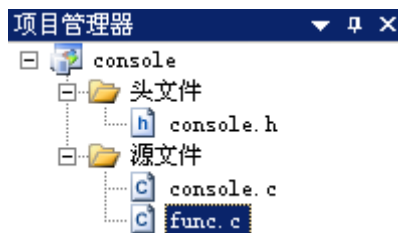


图 1-3: 添加 func.c 文件后的“项目管理器”窗口

5. 在 func.c 文件中添加函数：

```
int Func (int n)
{
    n = n + 1;
    return n;
}
```

6. 点击源代码编辑器上方的 console.c 标签，切换到 console.c 文件。将 main 函数修改为：

```
int main (int argc, char* argv[])
{
    int Func (int n); // 声明 Func 函数

    int n = 0;
    n = Func(10);
    printf ("Hello World!\n");
    return 0;
}
```

代码修改完毕后按 F7（“生成项目”功能的快捷键）。注意查看“输出”窗口中的内容，如果代码中存在语法错误，就根据错误信息进行修改，直到成功生成项目。

使用断点中断执行

1. 在 main 函数中定义变量 n 的代码行

```
int n = 0;
```

上点击鼠标右键，在弹出的快捷菜单中选择“插入/删除断点”，会在此行左侧的空白处显示一个红色圆点，表示已经成功在此行代码添加了一个断点，如图 1-4：

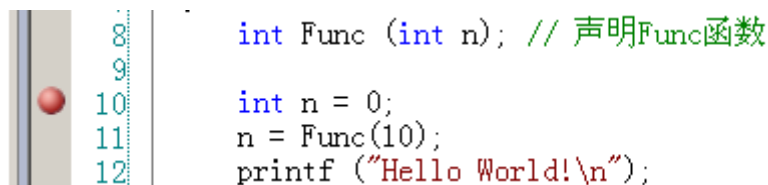


图 1-4: 在 console.c 文件的 main 函数中添加断点后的代码行

2. 在“调试”菜单中选择“启动调试”，Windows 控制台应用程序开始执行，随后 Linux Lab 窗口被自动激活，并且在刚刚添加断点的代码行左侧空白中显示一个黄色箭头，表示程序已经在此行代码处中断执行（也就是说下一个要执行的就是此行代码），如图 1-5:

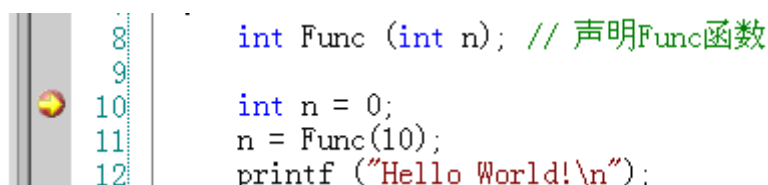


图 1-5: Windows 控制台应用程序启动调试后在断点处中断执行

3. 激活 Windows 控制台应用程序的窗口，可以看到窗口中没有输出任何内容，因为 printf 函数还没有被执行。

单步调试

按照下面的步骤练习使用“逐过程”功能：

1. 在 Linux Lab 的“调试”菜单中选择“逐过程”，“逐过程”功能会执行黄色箭头当前指向的代码行，并将黄色箭头指向下一个要执行的代码行。
2. 按 F10（“逐过程”功能的快捷键），黄色箭头就指向了调用 printf 函数的代码行。查看控制台应用程序窗口，仍然没有任何输出。
3. 再次按 F10 执行 printf 函数，查看控制台应用程序窗口，可以看到已经打印出了内容。
4. 在“调试”菜单中选择“停止调试”，结束此次调试。

按照下面的步骤练习使用“逐语句”功能和“跳出”功能：

1. 按 F5（“启动调试”功能的快捷键），仍然会在之前设置的断点处中断。
2. 按 F10 逐过程调试，此时黄色箭头指向了调用函数 Func 的代码行。
3. 在“调试”菜单中选择“逐语句”，可以发现黄色箭头指向了函数 Func 中，说明“逐语句”功能可以进入函数，从而调试函数中的语句。
4. 选择“调试”菜单中的“跳出”，会跳出 Func 函数，返回到上级函数中继续调试（此时 Func 函数已经执行完毕）。
5. 按 Shift+F5（“停止调试”功能的快捷键），结束此次调试。

查看变量的值

在调试的过程中，Linux Lab 提供了三种查看变量值的方法，按照下面的步骤练习这些方法：

1. 按 F5 启动调试，仍然会在之前设置的断点处中断。
2. 将鼠标移动到源代码编辑器中变量 n 的名称上，此时会弹出一个窗口显示出变量 n 当前的值（由于此时还没有给变量 n 赋值，所以是一个随机值）。

3. 在源代码编辑器中变量 `n` 的名称上点击鼠标右键，在弹出的快捷菜单中选择“快速监视”，可以使用“快速监视”对话框查看变量 `n` 的值。然后，可以点击“关闭”按钮关闭“快速监视”对话框。
4. 在源代码编辑器中变量 `n` 的名称上点击鼠标右键，在弹出的快捷菜单中选择“添加监视”，变量 `n` 就被添加到了“监视”窗口中。使用“监视”窗口可以随时查看变量的值和类型。此时按 F10 进行一次单步调试，可以看到“监视”窗口中变量 `n` 的值会变为 0，如图 1-6：

监视		
名称	值	类型
<code>n</code>	0x0	int

图 1-6：使用“监视”窗口查看变量的值和类型

如果需要使用十进制查看变量的值，可以点击工具栏上的“十六进制”按钮，从而在十六进制和十进制间切换。练习使用不同的进制和不同的方法来查看变量的值，然后结束此次调试。

调用堆栈

使用“调用堆栈”窗口可以在调试的过程中查看当前堆栈上的函数，还可以帮助理解函数的调用层次和调用过程。按照下面的步骤练习使用“调用堆栈”窗口：

1. 按 F5 启动调试，仍然会在之前设置的断点处中断。
2. 选择“调试”菜单“窗口”中的“调用堆栈”，激活“调用堆栈”窗口。可以看到当前“调用堆栈”窗口中只有一个 `main` 函数（显示的内容还包括了参数值和函数地址）。
3. 按 F11（“逐语句”功能的快捷键）调试，直到进入 `Func` 函数，查看“调用堆栈”窗口可以发现在堆栈上有两个函数 `Func` 和 `main`。其中当前正在调试的 `Func` 函数在栈顶位置，`main` 函数在栈底位置。说明是在 `main` 函数中调用了 `Func` 函数。
4. 在“调用堆栈”窗口中双击 `main` 函数所在的行，会有一个绿色箭头指向 `main` 函数所在的行，表示此函数是当前调用堆栈中的活动函数。同时，会将 `main` 函数所在的源代码文件打开，并也使用一个绿色箭头指向 `Func` 函数返回后的位置。
5. 在“调用堆栈”窗口中双击 `Func` 函数所在的行，可以重新激活此堆栈帧，并显示对应的源代码。
6. 反复双击“调用堆栈”窗口中 `Func` 函数和 `main` 函数所在的行，查看“监视”窗口中变量 `n` 的值，可以看到在不同的堆栈帧被激活时，Linux Lab 调试器会自动更新“监视”窗口中的数据，显示出对应于当前活动堆栈帧的信息。
7. 结束此次调试。

3.3 Linux 0.11 内核项目的生成和调试

之前练习了对 Windows 控制台应用程序项目的各项操作，对 Linux 0.11 内核项目的各种操作（包括新建、生成和各种调试功能等）与对 Windows 控制台项目的操作是完全一致的。所以，接下来实验内容的重点不再是各种操作的具体步骤，而应将注意力放在对 Linux 0.11 内核项目的理解上。

新建 Linux 0.11 内核项目

新建一个 Linux 0.11 内核项目的步骤如下：

1. 在“文件”菜单中选择“新建”，然后单击“项目”。
2. 在“新建项目”对话框中，选择项目模板“Linux011 Kernel”。
3. 在“名称”中输入新项目使用的文件夹名称“linux011”。
4. 在“位置”中输入新项目保存在磁盘上的位置“C:\”。
5. 点击“确定”按钮。

此项目就是一个 Linux 0.11 操作系统内核项目，包含了 Linux 0.11 操作系统内核的所有源代码文件。在“项目管理器”窗口中查看 Linux 0.11 内核项目包含的文件夹和源代码文件，可以看到不同的文件夹包含了 Linux 0.11 操作系统不同模块的源代码文件。也可以使用 Windows 资源管理器打开项目所在的文件夹 C:\linux011，查看所有源代码文件。

生成 Linux 0.11 内核项目

按 F7 生成项目，同时查看“输出”窗口中的内容，确认生成成功，如图 1-7：

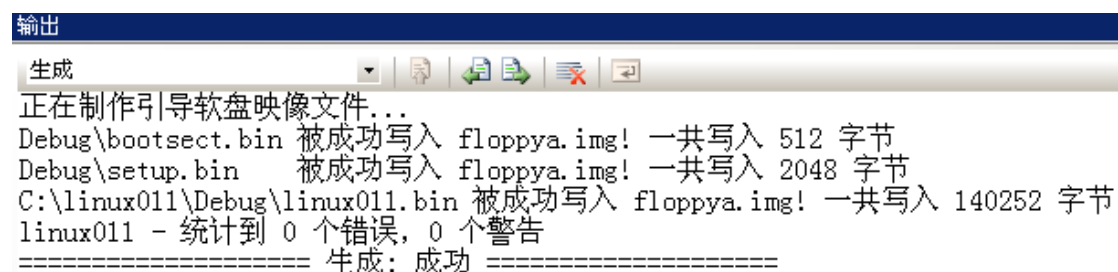


图 1-7: Linux 0.11 内核项目生成成功

打开 C:\linux011\debug 文件夹，可以查看刚刚生成的对象文件和目标文件。分别找到 bootsect.bin、setup.bin 和 linux011.bin 三个二进制文件，这三个二进制文件就是 Linux 0.11 操作系统需要运行的可执行文件。Linux Lab 在成功生成 Linux 0.11 操作系统内核后，会将这三个二进制文件写入大小为 1.44MB 的软盘镜像文件 floppy.a.img 中，并将该软盘镜像文件插入虚拟机的软盘驱动器 A 中，然后让虚拟机从软盘镜像 A 开始引导，并最终运行其中的 Linux 0.11 操作系统（相当于将写有三个二进制文件的软盘插入一个物理机的软盘驱动器 A 中，然后按下开机按钮）。

调试 Linux 0.11 内核项目

按照下面的步骤练习调试 Linux 0.11 源代码的过程：

1. 在“项目管理器”窗口的 init 文件夹中找到 main.c 文件节点，双击此文件节点使用源代码编辑器打开 main.c 文件。
2. 在 main.c 文件中 start 函数的“mem_init(main_memory_start, memory_end);”语句所在行（第 169 行）添加一个断点，如图 1-8 所示。Linux 0.11 启动时执行的第一个内核函数就是这个 start 函数。

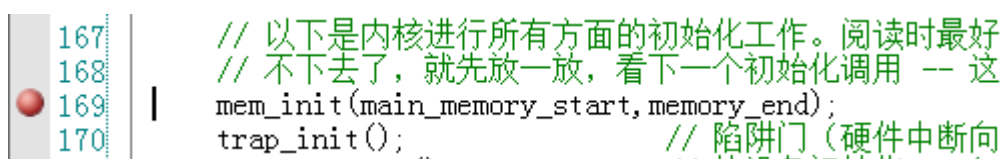


图 1-8: 在 Linux 0.11 内核项目的 init/main.c 文件中添加一个断点

3. 按 F5 启动调试，虚拟机开始运行软盘镜像中的 Linux 0.11。在虚拟机窗口中可以看到 Linux 0.11 启动的过程。随后 Linux 0.11 会在刚刚添加的断点处中断执行。

此时，激活虚拟机窗口可以看到 Linux 0.11 也不再继续运行了。各种调试功能（包括单步调试、查看变量的值和各个调试工具窗口）的使用方法与调试 Windows 控制台程序完全相同，读者可以在这里自己练习一下。

4. 按 F5 继续执行。此时查看虚拟机窗口，显示 Linux 0.11 操作系统已经启动，并且 Linux 0.11 的终端已经打开，可以接受用户输入的命令了。
5. 在“调试”菜单中选择“停止调试”，结束此次调试。

练习使用 Linux 0.11 的常用命令

在下面的表格中列出了 Linux 0.11 的常用命令。按 F5 启动 Linux 0.11，待终端打开后练习使用表格中的命令。练习完毕后，在“调试”菜单中选择“停止调试”，结束此次调试。

命令		命令功能	选项含义	
命令名	命令形式		选项	功能
ls		显示目录	-a	显示指定目录下的所有子目录与文件
			-d	如果参数是目录。就只显示其名称而不显示其包含的文件
			-R	显示指定目录的各个子目录中的文件
			-F	列出文件名后加不同符号，以区分文件类型
			-l	以长格式显示文件的详细信息。显示的信息依次为：文件类型与权限、链接数、文件属主、文件属组、文件大小、建立和最近修改的时间和文件名
cp	cp [选项] 源文件/目录 1 目标文件/目录 2	文件或目录复制	-a	复制时保留文件链接和属性，且复制所有子目录及其文件
			-r	若源文件为目录文件时，将复制该目录下所有子目录和文件。此时目标文件必须为目录名
mv		文件或目录移动		
rm	rm [选项] 文件列表	文件或目录删除	-r	删除参数中列出的全部目录及其子目录。如果没有使用 -r 选项，则不会删除目录
			-f	忽略不存在的文件，且不给出提示
mkdir		创建目录	-p	可以是一个路径名。此时若路径中的某些目录尚不存在，加此选项，系统将自动建立尚不存在的目录
rmdir		删除目录	-p	当子目录被完全删除时，父目录也被删除
pwd		显示工作目录路径		

cd		改变工作目录		
cat		显示文本文件的内容		
chmod	chmod [选项] [操作符][mode] 文件名	改变文件或目录访问权限	u	表示用户 (user)
			g	表示同组用户 (group)
			o	表示其他用户 (other)
			a	表示所有用户 (all), 系统的默认值
			操作符	+ (添加权限)、- (取消权限)、= (赋予权限, 并取消其他权限)
			mode	r (可读)、w (可写)、x (可执行)、u (与文件属主有相同的权限)、g (与文件属主同组的用户有相同的权限)、o (与其他用户有相同的权限)
exit		退出目前的 shell		
export		设置或显示环境变量		
sync		将主存缓冲区的数据写入磁盘		
df		查看文件空间的使用情况		
du		显示文件或目录占用文件空间情况		
echo		显示字符串		

注意:


- clear 命令用于清空屏幕, 但 Linux 0.11 没有实现, 可以使用 Ctrl+L 清空屏幕。
- 当修改磁盘数据后, 必须使用 sync 命令将主存缓存区的数据写入磁盘, 否则关闭虚拟机时修改的数据将会丢失。

练习使用查找功能

Linux 0.11 的源代码虽然不足两万行, 仅为任何一个具有商业价值的软件代码量的十分之一, 但是, 读者很可能还是首次接触到具有如此规模的源代码, 这为读者系统、高效的阅读和理解 Linux 0.11 的源代码带来不小的挑战。即便是一位有丰富经验的开发者, 在没有适当工具的帮助下, 阅读如此数量的源代码也会遇到很多困难。例如, 当遇到一个变量时, 想查看这个变量是在哪里定义的, 其数据类型是什么; 或者遇到一个函数时, 想查看这个函数是如何实现的, 在整个系统中都有哪些地方调用了此函数等问题。

Linux Lab 为了解决此类问题, 提供了强大而灵活的“查找”功能, 可用于在多个文件中快速、准确的查找文本, 从而帮助读者阅读和理解源代码。

请读者按照下面的步骤练习使用查找功能, 查找 Linux 0.11 内核源代码中对 fork 函数的声明、定义以及所有调用 fork 函数的代码:

1. 点击工具栏中的  按钮，或者按 Ctrl+Shift+F 快捷键，弹出“查找和替换”对话框，在“查找和替换”对话框的“查找内容”中输入“fork”，如图 1-9 所示。

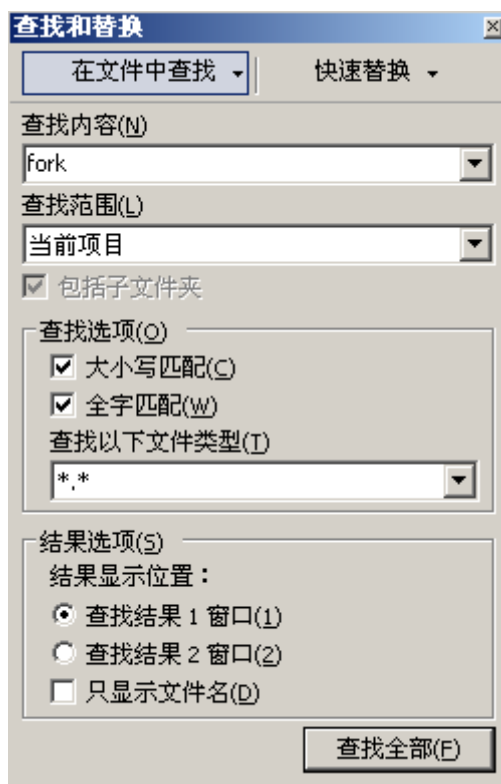


图 1-9: Linux Lab 提供的“查找和替换”对话框

2. 在“查找范围”下拉框中选择“当前项目”选项，如图 1-10 所示。

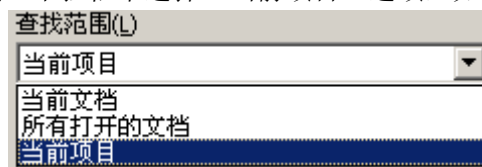


图 1-10: “查找范围”下拉框

3. 点击“查找全部”按钮，Linux Lab 会自动激活“查找结果”窗口，并会在其中显示出各个文件中出现此函数名称的位置。
4. 在“查找结果”窗口中双击要查看的位置，源代码编辑器会打开源代码文件，并将光标设置在文本所在行。这种方法还可以用来查找所有调用了 fork 函数的代码行，帮助读者理解该函数的使用方法。

这里需要注意的是，如果读者要查找的函数是在汇编代码中定义的，在 C 语言代码中被使用（或者相反），则函数的名称会不同（参见第 2.3 节），此时需要将“查找和替换”对话框中的“全字匹配”复选框取消选中。根据具体需要选择“大小写匹配”复选框是否选中。

练习使用快速查找功能

如果需要在某一文件中查找某一变量或函数的定义或者使用的位置，一种更简单的方法就是使用快速查找功能。具体操作方法为：用鼠标双击选中变量或者函数名，然后按 Ctrl+F3 快捷键，即可在该文件中向下查找该变量或者函数名。如果想向上查找，同样的方法按

Ctrl+Shift+F3 快捷键即可。

3.4 Linux 0.11 应用程序项目的生成和执行

新建 Linux 0.11 应用程序项目

新建一个 Linux 0.11 应用程序项目的步骤如下：

1. 在“文件”菜单中选择“新建”，然后单击“项目”。
2. 在“新建项目”对话框中，选择项目模板“Linux011 应用程序”。
3. 在“名称”中输入新项目使用的文件夹名称“linuxapp”。
4. 在“位置”中输入新项目保存在磁盘上的位置“C:\”。
5. 点击“确定”按钮。

此项目就是一个 Linux 0.11 应用程序项目。使用 Windows 资源管理器将之前生成的 C:\linux011\Floppya.img 文件拷贝覆盖 C:\linuxapp\Floppya.img 文件。这样 Linux 0.11 应用程序就可以使用最新版本的 Linux 0.11 操作系统内核了。

生成 Linux 0.11 应用程序项目

按 F7 生成项目，同时查看“输出”窗口中的内容，确认生成成功，如图 1-11：

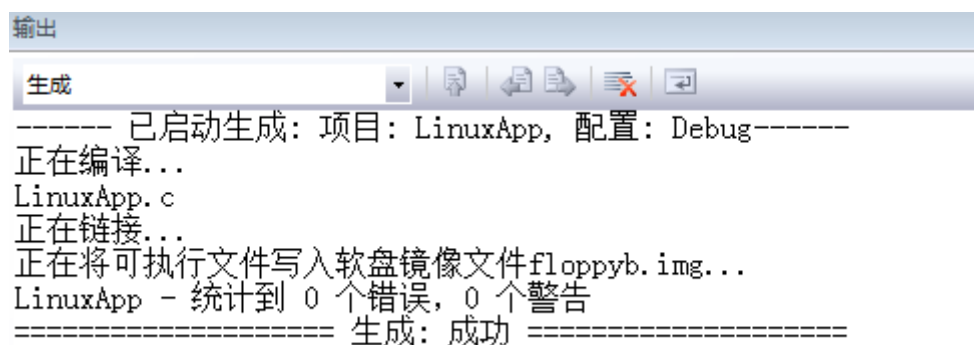


图 1-11：生成 Linux 0.11 应用程序项目

打开 C:\linuxapp\debug 文件夹，查看生成的对象文件和目标文件。其中的 linuxapp.exe 就是 Linux 0.11 应用程序的 aout 格式的可执行文件。注意，此可执行文件不能在 Windows 中运行，只能在 Linux 0.11 中运行。Linux Lab 每次生成应用程序项目时，都会将应用程序的可执行文件自动写入软盘镜像文件 floppyb.img 中。

查看软盘镜像文件中的内容

在“项目管理器”窗口中双击 floppyb.img 文件，使用 Floppy Image Editor 工具打开该项目中的 floppyb.img 文件，查看软盘镜像中的文件。linuxapp.exe 是从 C:\linux011app\debug 文件夹写入的，就是本项目生成的 Linux 0.11 应用程序，可以注意查看一下该文件的修改日期，如图 1-12：

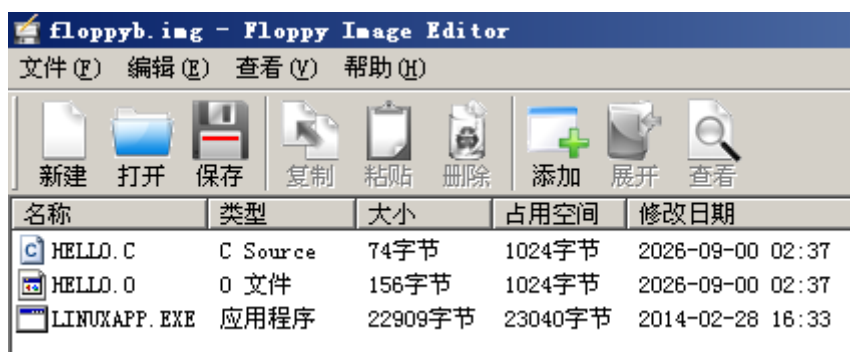


图 1-12: 查看 floppyb.img 中 Linux 0.11 应用程序项目生成的可执行文件

执行 Linux 0.11 应用程序项目

1. 按 F5 执行项目。
2. 待 Linux 0.11 启动后使用“mcopy b:linuxapp.exe linuxapp”命令将软盘镜像文件 floppyb.img 中的可执行文件 linuxapp.exe 拷贝到硬盘的当前目录中, 并命名为 linuxapp, 如图 1-13:

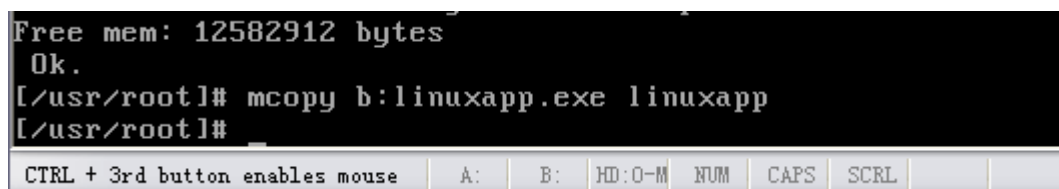


图 1-13: 将可执行文件 linuxapp.exe 拷贝至硬盘

3. 在终端输入“linuxapp”命令运行上一步拷贝的文件 linuxapp, 此时会提示“无法运行二进制文件”的信息, 原因是 linuxapp 文件不具有可执行的权限。使用“ls -l linuxapp”命令查看该文件的权限, 可以看到其只有 r 和 w 权限, 而没有 x 权限, 如图 1-14:

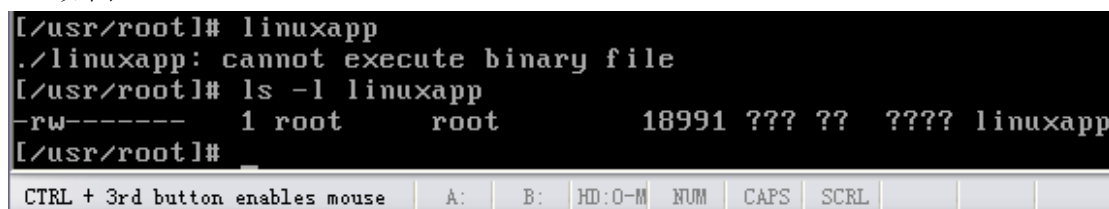


图 1-14: 使用 ls 命令查看文件 linuxapp 的权限

4. 使用“chmod +x linuxapp”命令使之具有可执行权限。再次使用 ls 命令确认文件权限修改后, 就可以运行该文件了, 如图 1-15:

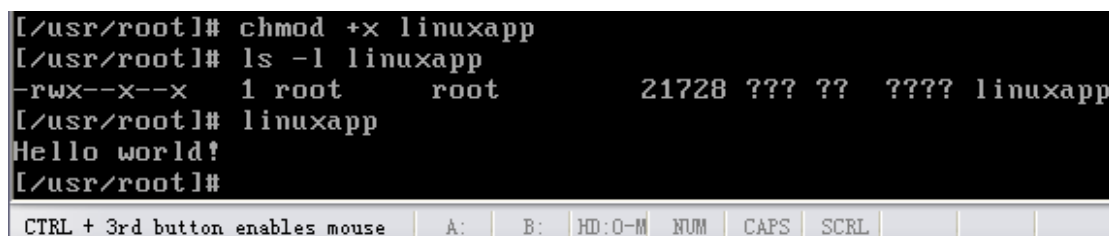


图 1-15: 为文件 linuxapp 增加可执行权限并运行

之前的内容详细介绍了使用 Linux Lab 提供的编辑器编写应用程序的源代码, 然后使用菜单中的“生成”功能自动产生应用程序可执行文件的过程。经常在 Windows 操作系统中

编写程序的读者会对这种操作方式十分熟悉，并觉得很方便。但是，在 Linux 0.11 操作系统中，由于没有可视化的人机交互界面，在其中编写应用程序的过程就会比较复杂，但是也有必要深入学习一下这种操作方式，为今后在 Linux 操作系统中编程打下基础。

接下来，会向读者详细介绍在 Linux 0.11 操作系统中如何使用 vi 编辑器编写源代码文件，如何使用 GCC 工具将源代码编译为可执行文件，以及如何编写 makefile 文件将多个源代码文件作为一个项目来进行管理的方法。

3.5 vi 编辑器的使用方法

vi (visual interpreter) 是 UNIX/Linux 操作系统使用的文本编辑器，是程序员编写程序的得力工具。vi 有 3 种操作模式：**命令模式**、**插入模式**和**末行模式**。

命令模式：当输入并执行 vi 命令后，会首先进入命令模式，此时输入的任何字符都被视为命令。命令模式用于控制屏幕光标移动、文本字符/字/行删除、移动复制某区段，以及进入插入模式或进入末行模式。

插入模式：在命令模式输入相应的插入命令（例如 i 命令）进入该模式。只有在插入模式下，才可以进行文字数据输入及添加代码，按 Esc 键可回到命令模式。

末行模式：在命令模式下输入某些特殊字符，如 “/”、“?” 和 “:”，才可进入末行模式。在该模式下可存储文件或退出编辑器，也可设置环境变量。

命令类型	命令形式	说明
进入 vi 命令	vi 文件名	显示 vi 编辑窗口, 载入指定的文件, 并进入命令模式
退出 vi 命令（退出 vi 时，若在插入模式，先按 Esc 返回命令模式）	:q!	放弃编辑内容，退出 vi
	:wq 或 :zz	保存文件，退出 vi
	:w	保存文件，但不退出 vi
	:q	退出 vi，若文件被修改过，要确认是否放弃修改的内容
进入末行模式（命令模式下，输入特殊字符进入末行模式）	:	进入末行命令模式
进入插入模式（命令模式下，执行下列命令均可进入插入模式）	i	插入命令
	a	附加命令
	o	打开命令
	s	替换命令
	c	修改命令
	r	取代命令

命令模式常用命令

命令	说明
x	删除光标所在的字符
X	删除光标所在位置前面的一个字符
nx	删除从光标开始到光标后 n-1 个字符
dw	删除光标到下一个单词起始位置
ndw	删除光标起的 n 个字
dd	删除光标所在的行

ndd	删除包括光标所在行的 n 行
Y	复制当前行至编辑缓冲区
nY	复制当前行开始的 n 行至编辑缓冲区
p	将编辑缓冲区的内容粘贴到光标的后面

3.6 使用 vi 和 GCC 编写 Linux 0.11 应用程序

接下来请读者按照下面的步骤，练习在 Linux 0.11 中使用 vi 编辑器编写源代码文件，然后使用 GCC 将源代码文件编译为可执行文件的过程。读者会感受到这种方法比较繁琐，没有使用 Linux lab 直接编写应用程序来的方便，不过学习这种方法还是很有必要的，当读者今后在一个只有命令终端的 Linux 系统上工作时，恐怕就只有使用这种方法来写程序了。

1. 按 F5 启动 Linux 0.11，使用 ls 命令查看当前目录，并用 rm 命令将 hello.c 文件删除。
2. 使用命令“vi hello.c”新建并打开 hello.c 文件，在此文件中使用 C 语言编写一个可以输出“hello world”的程序，保存源代码文件并退出 vi。
3. 使用 GCC 工具编译 hello.c，得到可执行文件 hello。命令如下：
gcc hello.c -o hello
4. 运行可执行文件 hello，查看输出的内容。

3.7 编写 Makefile 文件管理项目

Makefile 可以用来管理一个项目中多个源代码文件的编译和链接过程，也可以用来管理多个模块间的依赖关系，甚至是软件的安装过程。练习编写简单的 Makefile 文件，对于读者今后在 Linux 下开发程序还是非常必要的。

下面是一个可以完成编译链接 hello.c 任务的 makefile 文件的内容：

```
hello: hello.c
    gcc hello.c -o hello
```

第一行描述依赖关系，指出目标文件 hello 依赖于源代码文件 hello.c，第二行描述操作命令，指出要用 gcc 将源代码文件 hello.c 编译为可执行文件 hello。当修改 hello.c 文件后，需要编译时，只需在相同目录中输入 make 命令即可（注意，在 makefile 文件中跟在依赖关系命令行之后的操作命令行必须用制表符 (Tab 键) 缩进，否则会收到“missing separator”消息）。

也可以在 makefile 文件分步骤完成对 hello.c 文件的编译和链接，内容如下：

```
hello:hello.o
    gcc hello.o -o hello
hello.o:hello.c
    gcc -c hello.c -o hello.o
clean:
    rm -f *.o
    rm -f hello
```

可执行文件 hello 依赖于对象文件 hello.o，而 hello.o 又依赖于源代码文件 hello.c（选项 -c 指示 gcc 仅完成编译操作，而不进行链接操作）。另外又增加了一个 clean 目标，用于清理操作。可在任何时刻激活 clean 目标，删掉 makefile 生成的文件。

make 工具默认会构造 makefile 文件中的第一个目标，但如果在命令行中指定目标，就可以构建任何一个目标，例如指定 clean 目标的命令如下：

```
make clean
```

完成下面的练习：

1. 使用 vi 创建 add.c 文件，定义 int add(int, int)函数，计算并返回两个参数之和。
2. 使用 vi 创建 main.c 文件，定义 main 函数，在 main 函数中调用 add 函数，并打印输出计算的结果。
3. 使用 vi 创建 makefile 文件，分步完成对 add.c 和 main.c 文件的编译、链接、清理操作，可以生成一个可执行文件 main。

3.8 退出 Linux Lab

1. 在“文件”菜单中选择“退出”。
2. 在 Linux Lab 关闭前会弹出一个保存数据对话框，核对学号和姓名无误后点击“保存”按钮，Linux Lab 关闭。
3. 在 Linux Lab 关闭后默认会自动使用 Windows 资源管理器打开数据文件所在的文件夹，并且选中刚刚保存的数据文件（OUD 文件）。将数据文件备份（例如拷贝到自己的 U 盘中或者发送到服务器上），可以做为本次实验的考评依据。

3.9 学实验注意事项

1. 在 Linux 0.11 应用程序中不能使用“//”注释，只能使用“/*”和“*/”组合。
2. Linux Lab 不支持 Linux 0.11 应用程序的任何调试功能，即使在 Linux 0.11 应用程序源代码中添加了断点，调试过程无法命中断点。
3. Linux Lab 在生成 Linux 0.11 应用程序时，会自动将其可执行文件保存在软盘镜像文件 floppyb.img 中，启动 Linux 0.11 后需要使用 mcopy 命令将软盘 B 中的可执行文件复制到硬盘中才能运行。
4. 从软盘 B 复制到硬盘上的应用程序可执行文件默认是没有可执行权限的，需要使用 chmod 命令添加权限后才能执行。
5. 在 Linux 0.11 中对硬盘上的文件进行修改后，需要执行 sync 命令才能将修改保存在硬盘上，否则，关闭 Linux 0.11 后，修改会丢失。
6. Linux 0.11 还未实现 clear 命令，如需清理屏幕，可以使用 Ctrl+L 命令。

四、 思考与练习

1. 练习使用单步调试功能（逐过程、逐语句），体会在哪些情况下应该使用“逐过程”调试，在哪些情况下应该使用“逐语句”调试。练习使用各种调试工具（包括“监视”窗口、“调用堆栈”窗口等），为后续调试Linux内核做好准备。
2. 练习使用Linux 0.11提供的各种文件操作命令。使用这些命令浏览硬盘中的目录结构，了解Linux中各个目录的主要功能。

五、 相关阅读

1. 学习本书第 2 章，为后续阅读或者修改 Linux 0.11 的源代码做好准备。
2. 访问<http://www.engintime.com>了解关于Linux Lab的最新信息。登陆网站中的论坛，可以参与论坛中的讨论或者进行答疑。

实验二 操作系统的启动

实验性质：验证

建议学时：2 学时

实验难度：★★★★☆☆

一、实验目的

- 跟踪调试 Linux 0.11 在 PC 机上从 CPU 加电到完成初始化的过程。
- 查看 Linux 0.11 启动后的状态和行为，理解操作系统启动后的工作方式。

二、预备知识

现代操作系统启动比较复杂，涉及较多的名词，如：GDT、GDTR、IDT、A20 地址线、页表、保护模式等，有关这些介绍请读者自己查阅相关资料。本实验只介绍 Linux 0.11 操作系统启动的大致流程，不会关注太多的细节问题。

Linux 0.11 在 CPU 加电后、进入 `init/main.c` 中的 `start` 函数之前的阶段主要涉及三个源代码文件，分别是 `boot/bootsect.asm`、`boot/setup.asm` 和 `boot/head.s`。当处理器开始执行 `init/main.c` 中的 `start` 函数时，会完成系统的各项初始化工作，直到 Linux 完全启动，开始等待用户输入命令。现分别对上述所涉及的几个阶段做简要介绍。读者在阅读相关源代码时，可以结合这里介绍的总体流程来帮助理解，从而获得最佳的学习效果。

bootsect.asm

此文件使用 **NASM** 汇编语言编写，生成的二进制文件为 `bootsect.bin`。此二进制文件（512 字节）会被写入软盘镜像文件 `floppya.img` 的 0 号扇区，作为引导程序。在 CPU 加电完成 BIOS 自检程序的执行后，BIOS 程序会把软盘 A 的引导扇区加载到物理内存地址 `0x7c00` 处，并从该地址开始执行引导程序。引导程序会首先把自己移动到物理内存地址 `0x90000` 处，并继续执行。然后将软盘 A 的从 1 号扇区开始的 4 个扇区（即 `setup.bin`）加载到物理内存地址 `0x90200` 处。然后在屏幕上输出 “Loading system...”，并随后将扇区中的内核模块（即 `linux011.bin`）加载到物理内存地址 `0x10000` 处。最后，在确定根文件系统所在的磁盘后，跳转运行 `setup` 模块。

setup.asm

此文件使用 **NASM** 汇编语言编写，生成的二进制文件为 `setup.bin`。此二进制文件会被写入软盘镜像文件 `floppya.img` 的从 1 号扇区开始的 4 个扇区，作为操作系统加载程序。它首先利用 BIOS 中断读取机器参数，并放置在 `0x90000` 开始的物理地址处以备用。然后将内核模块从之前的 `0x10000` 起始处移动到 `0x00000` 起始处。随后加载中断描述符表寄存器（`idtr`）等，并在开启 A20 地址线后对 8259A 中断控制芯片重新编程。最后进入保护模式，跳转到地址 `0:0`（即内核模块中 `head.s` 的第一条指令处）运行。

head.s

此文件使用 **AT&T** 汇编语言编写，生成目标文件 `head.o`，此目标文件中的代码段会在链接时写入内核模块 `linux011.bin` 的代码段的开始位置。`head.s` 在执行时会首先加载各个段寄存器的值，重新设置中断描述符表 `idt`，并使各个表项（共 256 项）指向一个只报错误的哑中断子程序 `ignore_int`。最后 `head.s` 利用返回指令将预先放置在堆栈中的 `/init/main.c` 文件中的 `start` 函数的地址弹出，并跳转到 `start` 函数中去执行。

详细内容请读者阅读《Linux内核完全注释》第6章和第7章的内容。

三、 实验内容

3.1 准备实验

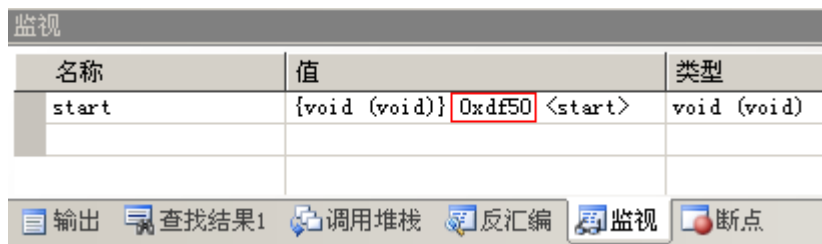
1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。
3. 在“项目管理器”窗口中打开 boot 文件夹中的 bootsect.asm 和 setup.asm 两个汇编文件。简单阅读一下这两个文件中的源代码和注释。
4. 按 F7 生成项目。
5. 生成完成后，使用 Windows 资源管理器打开项目文件夹中的 Debug 文件夹。找到由 bootsect.asm 生成的软盘引导扇区程序 bootsect.bin 文件，该文件一定为 512 字节（与软盘引导扇区的大小一致）。

3.2 调试 Linux 0.11 操作系统的启动过程

3.2.1 记录 init/main.c 文件中的内核入口函数(start 函数)的地址

首先按照下面的步骤记录下 start 函数的地址，留待后续使用：

1. 找到 start 函数的代码行(init/main.c 文件中的第 134 行)。
2. 在此代码行添加一个断点。
3. 按“F5”启动调试。
4. 会在刚刚添加的断点处中断。在 start 函数名称上点击右键，选择菜单中的“添加监视”，可以在监视窗口中查看 start 函数的地址，如图 2-1：



名称	值	类型
start	{void (void)} 0xdf50 <start>	void (void)

图 2-1:start 函数的地址

5. 请读者先在纸上记录图中红色圈出的函数地址。注意，读者得到的地址可能与图示中的不同，请读者记录实际的地址。
6. 选择“调试”菜单中的“结束调试”，结束此次调试。

3.2.2 使用 Bochs Debug 做为远程目标机

为了调试 BIOS 程序和软盘引导扇区程序，需要按照下面的步骤将远程目标机修改为 Bochs Debug：

1. 在“项目管理器”窗口中，右键点击项目节点，在弹出的快捷菜单中选择“属性”。
2. 在弹出的“属性页”对话框右侧的属性列表中找到“远程目标机”属性，将此属性值修改为“Bochs Debug”。
3. 点击“确定”按钮关闭“属性页”对话框。接下来就可以使用 Bochs 模拟器调试 BIOS 程序和软盘引导扇区程序了。

3.2.3 调试 BIOS 程序

按 F5 启动调试，此时会弹出两个 Bochs 窗口。标题为“Bochs for windows – Display”的窗口相当于计算机的显示器，显示操作系统的输出。标题为“Bochs for windows – Console”

的窗口是 Bochs 的控制台，用来输入调试命令和输出调试信息。

启动调试后，Bochs 在 CPU 要执行的第一条指令（即 BIOS 的第一条指令）处中断。此时，Display 窗口没有显示任何内容，Console 窗口显示将要执行的 BIOS 的第一条指令，并等待用户输入调试命令，如图 2-2：

```
[0xffffffff] f000:fff0 <unk. ctxt>: jmp far f000:e05b          ; ea5be000f0
<bochs:1> _
```

图 2-2: Console 窗口显示在 BIOS 第一条指令处中断

从 Console 窗口显示的内容中，可以获得关于 BIOS 的第一条指令的如下信息：

- 行首的[0xffffffff]表示此条指令所在的物理地址。
- f000:fff0 表示此条指令所在的逻辑地址（段地址:偏移地址）。
- jmp far f000:e05b 是此条指令的反汇编代码。
- 行末的 ea5be000f0 是此条指令的十六进制字节码，可以看出此条指令占用了 5 个字节。

接下来可以按照下面的步骤，查看 CPU 在没有执行任何指令前主要的寄存器中的数据（即 CPU 复位后的状态），以及内存中的数据：

1. 在 Console 窗口中输入调试命令 sreg 后按回车，显示当前 CPU 中各个段寄存器的值，如图 2-3。其中 CS 寄存器信息行中的“s=0xf000”表示 CS 寄存器的值为 0xf000。

```
<bochs:1> sreg
cs:s=0xf000, dl=0x0000ffff, dh=0xff0093ff, valid=1
ds:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
ss:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
es:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
fs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
gs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
ldtr:s=0x0000, dl=0x0000ffff, dh=0x00008200, valid=1
tr:s=0x0000, dl=0x0000ffff, dh=0x00008b00, valid=1
gdtr:base=0x00000000, limit=0xffff
idtr:base=0x00000000, limit=0xffff
<bochs:2>
```

图 2-3: 使用 sreg 查看寄存器中的值

2. 输入调试命令 r 后按回车，显示当前 CPU 中各个通用寄存器的值，如图 2-4。其中“rip: 0x00000000:0000ffff”表示 IP 寄存器的值为 0xffff。结合 BIOS 的第一条指令的地址，可以验证 CPU 将要执行的指令地址为 CS:IP。

```
<bochs:2> r
rax: 0x00000000:00000000 rcx: 0x00000000:00000000
rdx: 0x00000000:00000f20 rbx: 0x00000000:00000000
rsp: 0x00000000:00000000 rbp: 0x00000000:00000000
rsi: 0x00000000:00000000 rdi: 0x00000000:00000000
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000
r10: 0x00000000:00000000 r11: 0x00000000:00000000
r12: 0x00000000:00000000 r13: 0x00000000:00000000
r14: 0x00000000:00000000 r15: 0x00000000:00000000
rip: 0x00000000:0000ffff
eflags 0x00000002
id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af pf cf
<bochs:3>
```

图 2-4: 使用 r 寄存器查看寄存器中的值

3. 输入调试命令 xp /1024b 0x0000，查看从 0 开始的 1024 个字节的物理内存。在

Console 中输出的这 1K 物理内存的值都为 0，说明 BIOS 中断向量表还没有被加载到此处。

4. 输入调试命令 `xp /512b 0x7c00`，查看软盘引导扇区要被加载到的内存位置。输出的内存值都为 0，说明软盘引导扇区还没有被加载到此处。

通过以上的实验步骤，可以验证 BIOS 第一条指令的逻辑地址中的段地址和 CS 寄存器值是一致的，偏移地址和 IP 寄存器的值是一致的。由于内存还没有被使用，所以其中的值都为 0。

3.2.4 打开 lst 文件的方法（以 bootsect.asm 为例）

NASM 汇编器在将 `bootsect.asm` 生成 `bootsect.bin` 的同时，会生成一个 `bootsect.lst` 列表文件，此文件中的信息可以帮助开发者调试 `bootsect.asm` 文件中的汇编代码。

按照下面的步骤查看 `bootsect.lst` 文件中的内容：

1. 在 Linux Lab 的“项目管理器”窗口中打开“boot”文件夹，右键点击 `bootsect.asm` 文件。
2. 在弹出的快捷菜单中选择“打开生成的列表文件”，在源代码编辑器中就会打开文件 `bootsect.lst`。
3. 将 `bootsect.lst` 文件和 `bootsect.asm` 文件对比可以发现，此文件包含了 `bootsect.asm` 文件中所有的汇编代码，同时在代码的左侧又添加了更多的信息。
4. 在 `bootsect.lst` 中查找到软盘引导扇区程序第一条指令所在的行

```
32  00000000  B8C007  mov ax,BOOTSEG
```

此行包含的信息有：

- 32 是行号。
- 00000000 是此条指令相对于程序开始位置的偏移（第一条指令的偏移为 0）。
- B8C007 是此条指令的字节码，此条指令包含了 3 个字节。

3.2.5 调试软盘引导扇区程序（bootsect.asm）

BIOS 在执行完自检和初始化工作后，会将软盘引导扇区（512 字节）加载到物理地址 `0x7c00-0x7dff` 位置，并从 `0x7c00` 处的指令开始执行引导程序。按照下面的步骤从 `0x7c00` 处调试软盘引导扇区程序：

1. 输入调试命令 `vb 0x0000:0x7c00`，这样就在逻辑地址 `0x0000:0x7c00`（相当于物理地址 `0x7c00`）处添加了一个断点。
2. 输入调试命令 `c` 继续执行，会在 `0x7c00` 处的断点中断。中断后会在 Console 窗口中输出下一个要执行的指令，即软盘引导扇区程序的第一条指令，如图 2-5：

```
<0> [0x00007c00] 0000:7c00 <unk. ctxt>: mov ax, 0x07c0 ; b8c007
<bochs:7> _
```

图 2-5: 软盘引导扇区程序的第一条指令

3. 为了方便后面的使用，可以在纸上记录下此条指令的字节码（`b8c007`）。
4. 输入调试命令 `sreg` 验证 CS 寄存器（`0x0000`）的值
5. 输入调试命令 `r` 验证 IP 寄存器（`0x7c00`）的值。
6. 由于 BIOS 程序此时已经执行完毕，输入调试命令 `xp /1024b 0x0000` 验证此时 BIOS 中断向量表已经被载入。
7. 输入 `xp /512b 0x7c00` 显示软盘引导扇区程序的所有字节码。观察此块内存最开始的三个字节分别是 `0xb8`、`0xc0` 和 `0x07`，这和引导程序第一条指令的字节码是相同的。此块内存最后的两个字节分别是 `0x55` 和 `0xaa`，表示引导扇区是激活的，可以用来引导操作系统，这两个字节对应 `bootsect.asm` 中最后一行语句（注意，字节顺序使用 Little-endian）：


```
dw 0xAA55
```

接下来查看引导程序将自己复制到 0x90000 后的情况：

1. 输入调试命令 `xp /512b 0x9000:0x0000` 可以验证此时引导程序还没有将自己移动到 0x9000:0x0000 处。
2. 输入调试命令 `vb 0x9000:0x0018`，在 0x9000:0x0018 处设置一个断点。
3. 输入调试命令 `c` 继续执行，会在刚刚添加的断点处中断。
4. 再次输入调试命令 `xp /512b 0x9000:0x0000`，并与之前的内存比较，可以知道引导程序已经将自己移到了 0x90000 处，并在 0x9000:0x0018 处中断执行了。
5. 输入调试命令 `xp /512b 0x9000:0x0200`，可以验证此时 `setup.bin` 模块还没有被装入内存
6. 根据 `bootsect.lst` 文件下面一行的内容（执行此行指令时说明 `setup.bin` 加载完毕）

```
75 00000031 0F830B00 jnc ok_load_setup ; 读取成功
```

 输入调试命令 `vb 0x9000:0x0031`，在逻辑地址 0x9000:0x0031 处设置一个断点。
7. 输入调试命令 `c` 继续执行，会在刚刚添加的断点处中断。
8. 输入调试命令 `xp /512b 0x9000:0x0200`，此块内存已经发生改变，可以知道此时 `setup.bin` 模块已经被载入内存。
9. 输入调试命令 `xp /512b 0x1000:0x0000`，可以看到除前面有少量数据(BIOS 自检程序残留下)，后面全是 0。可以知道此时内核模块 `linux011.bin` 还没有被装载进入内存。
10. 根据 `bootsect.lst` 文件下面一行的内容（执行此行指令时说明 `linux011.bin` 加载完毕）

```
119 0000006E E8DB00 call kill_motor
```

 输入调试命令 `vb 0x9000:0x006e`。在 0x9000:0x006e 处设置一个断点。
11. 输入调试命令 `c` 继续执行，会在刚刚添加的断点处中断。
12. 输入调试命令 `xp /512b 0x1000:0x0000`，可以知道此时内核模块 `linux011.bin` 已经装载进入内存。

3.2.6 调试加载程序 (setup.asm)

`setup.asm` 生成的 `setup.bin` 被加载到从地址 0x90200 开始的物理内存。可以按照下面的步骤进行调试。

1. 输入调试命令 `vb 0x9020:0x0000`，在逻辑地址 0x9020:0x0000（即 `setup` 程序的第一条指令）处设置一个断点。
2. 输入调试命令 `c` 继续执行，可在 0x9020:0x0000 处中断，如图 2-6。打开 `setup.lst` 文件，可以看到其中第一条指令的字节码与此处将要执行的指令的字节码是相同的，说明 Linux 0.11 将要开始执行 `setup` 模块。

```
<0> [0x00090200] 9020:0000 (unk. ctxt): mov ax, 0x9000 ; b80090
<bochs:25>
```

图 2-6:在断点 0x9020:0x0000 处中断

3. 输入调试命令 `xp /2b 0x9000:0x0000`，查看取得各种机器参数之前物理内存 0x90000-0x901FF 中前两个字节的值并记录下来，如图 2-7。

```
<bochs:25> xp /2b 0x9000:0x0000
[bochs]:
0x00000000000090000 <bogus+ 0>: 0xb8 0xc0
<bochs:26>
```

图 2-7:取得机器参数前物理内存 0x90000 处的值。

4. 根据 `setup.lst` 文件下面一行的内容


```
116 00000080 FA cli
```

输入调试命令 `vb 0x9020:0x0080`

5. 输入调试命令 `c` 继续运行，在刚刚添加的断点处中断。此时 Linux 0.11 已经将各种机器参数放入 `0x90000-0x901FF` 的物理内存中。
6. 输入调试命令 `xp /2b 0x9000:0x0000`，查看此处内存的值，如图 2-8，并和步骤 3 进行对比。可以知道此时该地址处的内存值已经改变。

```
<bochs:28> xp /2b 0x9000:0x0000
[bochs]:
0x0000000000000090000 <bogus+ 0>: 0x00 0x15
<bochs:29>
```

图 2-8:取得机器参数后 `0x90000` 处内存的值

7. 根据 `setup.lst` 文件下面一行的内容

```
209 0000010B EA00000800 jmp 8:0
```

输入调试命令 `vb 0x9020:0x010b`

8. 输入调试命令 `c` 继续执行。可以在步骤 7 设置的断点处停下。此时，下一条将要执行的指令会跳转到 `head.s` 的第一条指令处执行。

3.2.7 调试内核模块中的 `head.s`

1. 打开 `head.lst`，找到其第一条指令所在行，如下（此条指令包含了 5 个字节）：

```
25 0000 B8100000 movl $0x10,%eax
25 00
```

在生成过程中，`head.s` 会生成目标文件 `head.o`，此目标文件中的代码段在链接时会写入 `linux011.bin` 中代码段的开始位置，并且 `linux011.bin` 会被加载到从 `0x0000` 开始的物理内存。所以，可以在物理地址 `0x0000` 处设置一个断点（注意执行到这里时，CPU 已经处于保护模式了），输入的调试命令为：`pb 0x0000`。

2. 输入调试命令 `c` 继续执行，可在图 2-9 所示的指令处中断。

```
<0> [0x00000000] 0008:0000000000000000 <unk. ctxt>: mov eax, 0x00000010
b810000000
<bochs:33>
```

图 2-9:在 `head.s` 的第一条指令处中断

对比 `head.lst` 文件中第一条指令的字节码，可以确认已经进入了 `head.s` 模块。

3. 根据 `head.lst` 文件下面一行的内容

```
202 540b 68000000 pushl $_start
202 00
```

输入调试命令：`pb 0x540b`。该指令负责将内核入口点函数 `start` 的地址压入堆栈。

4. 输入调试命令 `c` 继续执行，在步骤 3 所设置的断点处中断，如图 2-10。

```
<0> Breakpoint 1, 0x000000000000540b in ?? (<)
Next at t=34657312
<0> [0x0000540b] 0008:000000000000540b <unk. ctxt>: push 0x0000df50
6850df0000
<bochs:3>
```

图 2-10:在将 `start` 函数地址压栈的指令处中断

将图中红线中的地址与 3.2.1 中记录的 `start` 函数的地址比较，可以确认这个地址就是 `start` 函数的地址。最终就是通过这个地址跳转到操作系统内核的入口点函数开始执行，从而结束引导过程的。

5. 根据 `head.lst` 文件下面一行的内容

```
316      54a5    C3      ret
```

输入调试命令: `pb 0x54a5`。

6. 输入调试命令 `c` 继续执行, 在 `ret` 指令处中断。然后接着输入调试命令 `s`, 单步执行此行的 `ret` 指令, 可以得到图 2-11:

```
(0) Breakpoint 2, 0x00000000000054a5 in ?? (<)>
Next at t=34674738
(0) [0x000054a5] 0008:00000000000054a5 (unk. ctxt): ret
c3
<bochs:5> s
Next at t=34674739
(0) [0x0000df50] 0008:000000000000df50 (unk. ctxt): push ebp
55
<bochs:6>
```

图 2-11: 准备执行 `start` 函数的第一条指令

图中红线圈出的是 IP 寄存器将要执行的下一条指令。对比可以发现它和图 2-10 中圈出的值是一模一样的, 说明接下来就开始执行 `start` 函数中的指令了。将内核入口点函数 `start` 的地址压入堆栈, 然后再通过 `ret` 指令进入此函数执行是这里的一个小技巧, 请读者仔细体会。

7. `head.s` 调试到这里, `init/main.c` 的 `start` 函数之前部分就全部结束了, 系统将跳入 `init/main.c` 中的 `start` 函数中去执行。在 `head.s` 中还做了许多其它事情, 如对中断描述符表和全局描述符表进行设置, 对页表的设置等, 有兴趣的读者可以自己进行验证。

3.3 内核初始化

通过对比图 2-1 和图 2-11 中的函数地址, 可以知道, 处理器最终跳转到文件 `init/main.c` 的 `start` 函数中去执行。说明此时已经进入了内核, 但从刚进入内核到系统最终稳定下来, 等待用户输入命令, 还有很多初始化工作要做。由于此过程较为复杂, 不适合刚刚接触操作系统的读者学习, 这里只做总体介绍, 如有读者想了解初始化的详细过程, 请阅读《Linux 内核完全注释》第 7 章的内容。

读者可以在系统完成启动后, 在终端的命令行按 `F1` 键, 会显示如下的信息:

```
[usr/root]# 0: pid=0, state=1, 2727 (of 3140) chars free in kernel stack
1: pid=1, state=1, 2492 (of 3140) chars free in kernel stack
2: pid=4, state=1, 1440 (of 3140) chars free in kernel stack
3: pid=3, state=1, 1428 (of 3140) chars free in kernel stack
A
```

图 2-12 Linux 0.11 刚启动后的系统进程信息

显示系统中存在四个进程, 进程号分别为 0、1、4、3。进程号为 0 的进程是 Linux 引导程序中创建的第一个进程, 在完成加载系统后, 演变为进程调度, 交换以及存储管理进程。然后, 以进程 0 为母本创建进程 1, 来继续完成系统的初始化, 进程 1 是系统中所有其它用户进程的祖先进程。然后, 以进程 1 为母本创建进程号为 2 的 Shell 进程。Shell 进程再创建进程号为 3 的 `update` 进程, 然后进程 2 就退出了, 所以在列表中没有显示 2 号进程。`update` 进程的主要任务是将缓冲区中的数据同步到外设(软盘和硬盘)。`update` 进程启动后将会被挂起, 然后让进程 1 继续运行, 此时进程 1 会重建 Shell 进程接受用户的命令, 其进程号为 4。

3.4 mkfloppy 和 pe2bin 应用程序

在 Linux Lab 的“新建项目”对话框中可以选择 `mkfloppy` 项目模板或 `pe2bin` 项目模板来创建 Windows 控制台应用程序。在 Linux 0.11 内核项目的生成过程中会调用这两个应用程序, 来生成 Linux 0.11 内核的可执行文件 `linux011.bin`。虽然在安装 Linux Lab 时, 已经将这两个应用程序的可执行文件 `mkfloppy.exe` 和 `pe2bin.exe` 安装到了 Linux lab 的 `bin` 文件夹中, 但是

读者还是有必要学习一下这两个应用程序的源代码，甚至是修改它们的源代码来实现一些特殊的功能，从而加深对 Linux 0.11 系统的理解。

mkfloppy 项目模板创建的应用程序 mkfloppy.exe 用于将 bootsect.bin、setup.bin 和 linux011.bin 三个文件写入软盘映像文件 floppy.img 中。一个软盘扇区的大小是 512 字节，bootsect.bin 文件占用软盘的第 0 扇区，setup.bin 文件占用软盘的第 1-4 扇区，linux011.bin 文件占用软盘的第 5-388 扇区。

pe2bin 项目模板创建的应用程序 pe2bin.exe 用于将生成的 PE 格式的可执行文件 linux011.exe 转换成平坦的 linux011.bin 文件。PE 格式的可执行文件 linux011.exe 中存放着 Linux 0.11 操作系统内核的数据和指令以及调试信息，由于其格式比较复杂，还无法被 bootsect 程序识别，所以无法直接从软盘加载到内存中，所以，pe2bin.exe 应用程序的目的就是将 linux011.exe 文件中的数据 and 指令信息抽取出来，并重新保存成 linux011.bin 文件，从而允许 bootsect 程序直接将其加载到内存中。

在 linux 0.11 内核项目中，在左侧的项目管理器中右击 linux011 节点，单击弹出菜单中的“属性”，将会弹出“属性页”对话框，单击左侧树中的“生成后事件”节点，在右侧的“命令行”属性中会显示如下内容：

```
echo 正在生成内核映像文件...
copy "$(TargetPath)" "$(TargetDir)$$(TargetName).tmp"
strip "$(TargetDir)$$(TargetName).tmp"
pe2bin.exe "$(TargetDir)$$(TargetName).tmp" "$(TargetDir)$$(TargetName).bin"
del "$(TargetDir)$$(TargetName).tmp"
echo 正在制作引导软盘映像文件...
mkfloppy.exe "$(OutDir)\bootsect.bin" "$(OutDir)\setup.bin" "$(TargetDir)$$(TargetName).bin"
"floppy.img"
```

这些 Windows 控制台命令会在生成 Linux 0.11 内核项目的最后阶段自动执行。其中 echo 命令用于打印信息；copy 命令将 linux011.exe 文件拷贝成一个 linux011.tmp 临时文件；strip 命令将 linux011.tmp 文件中的调试信息删除；pe2bin.exe 将 linux011.tmp 文件转换为 linux011.bin 文件；del 命令将 linux011.tmp 临时文件删除；mkfloppy.exe 将 bootsect.bin、setup.bin 和 linux011.bin 文件写入到 floppy.img 软盘镜像文件中。

四、 思考与练习

1. 为什么Linux 0.11操作系统从软盘启动时要使用bootsect.bin和setup.bin两个程序？使用一个可以吗？它们各自的主要功能是什么？如果将setup.bin的功能移动到bootsect.bin文件中，则bootsect.bin文件的大小是否仍然能保持小于512字节？
2. 结合前面“预备知识”中关于三个源文件介绍的总体流程，在源代码文件中找到相关实现代码，加深对操作系统启动过程的理解。
3. 使用 mkfloppy 模板新建一个应用程序项目，仔细阅读其中的源代码。此应用程序默认会将 linux011.bin 文件写入软盘镜像文件从 5 号扇区开始的位置，尝试修改该应用程序将 linux011.bin 文件写入从 8 号扇区开始的位置。完成后按 F7 生成项目，将生成的 mkfloppy.exe 拷贝替换 Linux lab 安装目录 bin 文件夹下的 mkfloppy.exe，修改 boot\bootsect.asm 中的汇编代码，使之能够从软盘 A 的 8 号扇区加载 linux011.bin 文件，确保 Linux 0.11 能够正常启动。
4. 将Linux 0.11内核项目的“生成后事件”中的命令行修改为如下：

```
echo 正在生成内核映像文件...
strip "$(TargetPath) "
```

echo 正在制作引导软盘映像文件...

```
mkfloppy.exe "$(OutDir)\bootsect.bin" "$(OutDir)\setup.bin" "$(TargetPath)" "floppya.img"
```

这样就会将 PE 格式的内核文件 linux011.exe 直接写入软盘镜像文件中，这就需要读者修改 boot\bootsect.asm 中的汇编代码，使之在读取软盘 A 中的内核文件时，能够将 PE 文件中的指令和数据直接加在到内存中适当的位置，从而使 Linux 0.11 能够正常启动。可以参考 pe2bin 项目模板中的源代码完成此练习。

五、 相关阅读

1. 打开Linux Lab的“帮助”菜单，选择“其它帮助文档”中的“NASM手册”，阅读该手册了解NASM汇编器的特点。也可以访问NASM的网站<http://www.nasm.us>了解最新的信息。
2. 打开Linux Lab的“帮助”菜单，选择“其它帮助文档”中的“Intel 80386编程手册”，阅读该手册中的17.2.2.11节可以查看各个汇编指令的详细信息。关于CPU初始时状态，可以参看第10章的10.1和10.2节。阅读第14章可以了解更多关于实模式的信息。

实验三 Shell 程序设计

实验性质：验证、设计

建议学时：2 学时

实验难度：★★★★☆☆

一、实验目的

- 了解 Shell 在 Linux 中的重要作用。
- 学会编写简单的 Shell 脚本程序。

二、预备知识

Shell 为用户和 Linux 操作系统之间的命令交互提供最基本的接口，在使用者和操作系统之间架起一座桥梁。它的名字 Shell(外壳)形象地表示它在用户与操作系统之间的关系。早期 Shell 主要用作命令解释器，经过不断扩充和发展，现在已是命令语言、命令解释器、程序设计语言的统称，被泛指为一个提供人机交互界面和接口的程序。

Shell 是一种解释型程序设计语言，它支持大多数高级程序设计语言中能见到的程序结构，如函数、变量和控制语句，具有极强的程序设计能力，可被用来编写 Shell 脚本。Shell 脚本包含一系列命令，运行脚本就是执行脚本中的每个命令，可用一个脚本在一次运行中执行许多个命令。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。

3.2 回显语句

echo 语句用来显示变量值或字符串。

用法：echo 变量值|字符串

1. 使用 vi 新建 Shell 脚本文件 Shell1.sh。
2. 编辑 Shell1.sh 的内容如下（‘#’后的语句为注释）：
#print hello world
echo hello world
3. 保存文件后退出 vi。用 chmod 命令使 Shell1.sh 具有可执行权限。
4. 在 Shell 中输入 Shell1.sh 后回车，观察运行结果。

3.3 特殊字符

字符	作用
星号*	通配符，可匹配任何字符串
问号?	通配符，可匹配任何单个字符
方括号[]	匹配括号内所有字符中的任一字符，可在括号内的字符间用短划线“-”表示字符范围，例如，myfile[23456]与 myfile[2-6]效果相同，可以匹配文件 myfile2 至 myfile6。
反斜杠\	为了将 Shell 的特殊字符*、?、[]、&和;等变成普通字符，必须在这些字符前加\，此外，\放行尾为续行符。

双引号""	其所括字符中，除\$、\及,外的其他特殊字符都失去特定含义。
单引号''	其所括字符中的所有特殊字符都失去特定含义。
倒引号``	其所括字符串被 Shell 解析为命令行。

编写如下 Shell 脚本文件 Shell2.sh，并观察运行结果：

```
rm -f *.o
rm -f hell?.c
echo `pwd`
```

3.4 变量

变量是用于保存值或正文的词，变量可以被创建、赋值和删除，可以现定义、现赋值。通常所有变量都被看做字符串并以字符串来存储，即使它们被赋值为整数时也是如此。Shell 和软件工具会在需要时把数值型字符串转换为对应的数值以对它们进行操作。

1. 用户自定义变量

用户自定义变量也称局部变量，是用户在 Shell 脚本中定义的变量，可以对它赋值，可以是整形值、字符串(包含在双引号中)等。引用变量时，需要在其名称前面添加“\$”。定义变量的格式为：

变量名=字符串

编写如下 Shell 脚本文件 Shell3.sh，并观察运行结果：

```
a=1
echo $a
b="hello world"
echo $b
```

2. 位置变量

位置变量类似于 C 语言程序的命令行参数，可以写在 Shell 程序文件名之后，共有 9 个，用 \$n 表示 (n 为一个十进制数)。在参数传递时必须使命令行中提供的参数与程序中的位置参数一一对应。\$0 是一个特殊变量，它是当前 Shell 程序的文件名，第 1 个位置变量名为 \$1，第 2 个位置变量名为 \$2，以此类推。

编写如下 Shell 脚本文件 Shell4.sh，然后在命令行中输入“Shell4.sh 100 “hello””，并观察运行结果：

```
echo $0
echo $1
echo $2
```

3. Shell 变量

Shell 变量的名字和含义是固定的。\$?为得到上次命令的十进制返回码，\$\$为得到当前 Shell 进程的进程号，\$!为得到上个后台进程的进程号，\$#为得到传递给 Shell 程序的位置参数个数（不限于 9 个，但不包含 Shell 文件名），\$-为当前 Shell（用 set）设置的执行标识名组成的字符串，\$*为命令行中实际给出的实参字符串。Shell 变量是由 Shell 程序本身设置的特殊变量，根据实际情况设置，不允许用户重新设置。

编写如下 Shell 脚本文件 Shell5.sh，并观察运行结果：

```
echo $$
echo $#
```

4. 命令替换

一些变量的值取决于用一对倒引号（`）括起来的命令执行的结果。

编写如下 Shell 脚本文件 Shell6.sh，并观察运行结果：

```
curr=`pwd`
ll=`ls -l`
echo $curr
echo $ll
```

注意：变量名只能包含大、小写字母、数字（0~9）和下划线，且只能以字母或下划线开头。

3.5 运算语句

let 语句后可跟算术表达式执行整数算数运算，因而 Shell 含有各种算数运算符。

编写如下 Shell 脚本文件 Shell7.sh，并观察运行结果：

```
i=1
j=2
let k=$i+$j
echo $k
```

3.6 函数

Shell 脚本函数与一般语言中函数的用法类似，必须先定义后使用，并且可以传递参数。

定义

```
function Fun()
{
    函数体
}
```

返回值

Shell 脚本函数的返回值可以通过 return 关键字返回，返回值只能为整数或整数型数值字符串，也可以无返回值。

需要注意的是，Shell 脚本函数的返回值不能通过 ‘=’ 直接赋值给变量，只能在调用函数后，通过 ‘\$?’ 来获得函数的返回值，且中间不能有其它语句。

参数传递

Shell 脚本函数的参数传递与 Shell 脚本的位置变量类似，也是通过 \$n 表示，n 是一个十进制数。

调用

在 Shell 脚本中调用函数与执行命令的格式一样：函数名 参数列表

编写如下 Shell 脚本文件 Shell8.sh，并观察运行结果：

```
function Mul()
{
    let i=$1*$2
    return $i
}
Mul 2 3
a=$?
echo $a
```

3.7 控制语句

命令在脚本中的执行顺序称为脚本流。当编写一个较复杂的脚本时，常常需要根据不同的条件执行不同的脚本流，这就要求 Shell 提供各种流程控制语句。

1. 测试语句

测试语句通常作为条件表达式来使用，通过测试表达式 `expression` 来实现一个条件测试。测试语句计算表达式的值，并返回真(0)或假(1)。其在 Shell 脚本中常常缩写为 `[expression]` 的形式（注意，中括号与表达式之间必须有空格）。

文件测试

测试条件	返回值说明
<code>[-r file]</code>	表示若是文件存在且用户可读，则测试条件为真
<code>[-w file]</code>	表示若文件存在且用户可写，则测试条件为真
<code>[-x file]</code>	表示若文件存在且用户可执行，则测试条件为真
<code>[-b file]</code>	表示若文件存在且为块设备文件，则测试条件为真
<code>[-c file]</code>	表示若文件存在且为字符设备文件，则测试条件为真
<code>[-d file]</code>	表示若文件存在且为目录文件，则测试条件为真
<code>[-f file]</code>	表示若文件存在且为普通文件，则测试条件为真
<code>[-p file]</code>	表示若文件存在且为 FIFO 文件，则测试条件为真
<code>[-s file]</code>	表示若文件存在文件长度>0，则测试条件为真
<code>[-t file]</code>	表示若文件描述符与终端相关，则测试条件为真

字符串比较

测试条件	返回值说明
<code>[-z s1]</code>	表示若字符串长度为 0，则测试条件为真
<code>[-n s1]</code>	表示若字符串长度>0，则测试条件为真
<code>[s1]</code>	表示若 <code>s1</code> 不是空字符串，则测试条件为真
<code>[s1=s2]</code>	表示若两个字符串相等，则测试条件为真
<code>[s1!=s2]</code>	表示若两个字符串不相等，则测试条件为真
<code>[s1<s2]</code>	表示若按字典顺序 <code>s1</code> 在 <code>s2</code> 之前，则测试条件为真
<code>[s1>s2]</code>	表示若按字典顺序 <code>s1</code> 在 <code>s2</code> 之后，则测试条件为真

整数比较

测试条件	返回值说明
<code>[int1 -gt int2]</code>	表示若 <code>int1</code> 大于 <code>int2</code> ，则测试条件为真
<code>[int1 -eq int2]</code>	表示若 <code>int1</code> 等于 <code>int2</code> ，则测试条件为真
<code>[int1 -ne int2]</code>	表示若 <code>int1</code> 不等于 <code>int2</code> ，则测试条件为真
<code>[int1 -lt int2]</code>	表示若 <code>int1</code> 小于 <code>int2</code> ，则测试条件为真
<code>[int1 -le int2]</code>	表示若 <code>int1</code> 小于或等于 <code>int2</code> ，则测试条件为真
<code>[int1 -ge int2]</code>	表示若 <code>int1</code> 大于或等于 <code>int2</code> ，则测试条件为真

2. 判断语句

格式为：

if 条件表达式 then 命令 else 命令 fi

条件表达式为任意逻辑表达式，大多数返回一个整数值。返回 0 时执行 `then` 后的语句，否则执行 `else` 后的语句。条件表达式也可以使用两个预定义的值 `true` 和 `false`。

编写如下 Shell 脚本文件 `Shell9.sh`，用于判断一个普通文件是否存在。然后在控制台中输入命令“`shell9.sh shell9.sh`”，观察运行结果。

```
if [ -f $1 ]
then echo "File exist."
else echo "Can not find file."
```


fi

3. 开关语句

格式为：

case 字符串 in 模式字符串 1) 命令表 1;; ... 模式字符串 n) 命令表 n;; esac

case 允许多重条件选择，执行过程是：用字符串去匹配各模式字符串，发现某个匹配，便执行其后面的语句。

编写如下 Shell 脚本文件 Shell10.sh，用于判断传递给 Shell 脚本的参数个数，在命令行中输入不同数量的参数，观察运行结果：

```
case $# in
0) echo "0 argument";;
1) echo "1 argument";;
2) echo "2 arguments";;
*) echo "more than 2";;
esac
```

4. 循环语句

循环语句分三种：while、for 和 until。

While 语句格式为：

while 测试命令 do 命令表 done

若测试命令返回 true，则进入循环体执行命令表，然后再执行测试命令，直至其返回 false 为止。

编写如下 Shell 脚本文件 Shell11.sh，观察运行结果：

```
i=0
while [ $i -lt 6 ]
do
echo $i
let i=$i+1
done
```

注意：[] 内的测试命令两端都要各加一个空格！

for 语句格式为：

for 变量 in 值表 do 命令表 done

变量依次取值表中的各个值，然后进入循环体并执行命令表中的命令，变量变为空时结束 for 循环。

编写如下 Shell 脚本文件 Shell12.sh，观察运行结果：

```
for i in 0 1 2 3 4 5
do
echo $i
done
```

until 语句格式为：

until 测试命令 do 命令表 done

与 while 语句相似，但当测试条件为 false 时，进入循环体执行，直至测试条件为 true 时结束 until 语句。请读者编写一个 Shell 脚本文件，使用 until 循环语句打印数字 0-5。

请读者完成下面的练习：

- a) 编写如下 Shell 脚本文件 Shell13.sh，使用 while 循环语句打印用字母 a 填充的三角形：

```
i=1
while [ $i -lt 6 ]
do
    j=0
    str=""
    while [ $j -lt $i ]
    do
        str="$str `echo a`"
        let j=$j+1
    done
    echo $str
    let i=$i+1
done
```

运行结果如下：

```
a
a a
a a a
a a a a
a a a a a
```

请分别使用 for 语句和 until 语句编写具有类似功能的 Shell 脚本文件。

- b) 请选择一种循环语句编写 Shell 脚本文件，实现打印九九乘法表功能。

5. 读入语句

read 语句从标准输入（键盘）上读取数据行，并给局部变量赋值。

编写如下 Shell 脚本文件 Shell14.sh，从键盘读取数据并赋给变量 a，观察运行结果：

```
echo "please input data"
read a
echo $a
```

6. 退出语句

break 语句可以退出循环体，continue 语句可以返回本层循环头，然后开始新一轮循环。

break[n] 表示跳出 n 层循环，默认为 1。continue[n] 语句表示退出到包含本语句的第 n 层，然后继续循环。

7. 退出程序语句

exit 语句可以退出正在执行的 Shell 脚本。

编写如下 Shell 脚本文件 Shell15.sh，并观察运行结果：

```
for i in 1 2 3
do
    echo $i
    break
```

```
done  
exit  
echo $i
```

8. 等待语句

`wait` 语句使 Shell 等待后台启动的所有子进程结束后再继续运行。

四、思考与练习

1. 编写一个 Shell 脚本文件 `Calculator.sh`，要求从命令行输入简单的算术表达式(加减乘除)，根据不同的运算符调用不同的函数计算并返回结果，最后打印结果。例如输入命令：
`Calculator.sh 2 + 4`
将调用计算加法的函数后返回结果 6，并打印“2+4=6”。
2. 编写一个 Shell 脚本文件，要求执行时，如果第一个位置参数是合法的目录，那么就把该目录及其所有子目录下的文件名称显示出来，如果第一个位置参数不是合法的目录，则显示目录名不对。（提示：脚本函数也可以使用递归）

实验四 系统调用

实验性质：验证

建议学时：2 学时

实验难度：★★★☆☆

一、实验目的

- 深入了解 Linux 系统调用的执行过程，建立对系统调用的深入认识。
- 学会增加系统调用及添加内核函数的方法。

二、预备知识

系统调用是操作系统为应用程序提供的与内核进行交互的一组接口。通过这些接口，用户态应用程序的进程可以切换到内核态，由系统调用对应的内核函数代表该进程继续运行，从而可以访问操作系统维护的各种资源，实现应用程序与内核的交互。

系统调用通过中断机制向内核提交请求，系统调用的功能由内核函数实现，进入内核后不同的系统调用会有其各自对应的内核函数，这些内核函数就是系统调用的“服务例程”。

Linux 内核为了区分不同的系统调用，为每个系统调用分配了唯一的系统调用号。这些系统调用号定义在文件 `include/unistd.h` 中，系统调用号实际上对应于 `include/linux/sys.h` 中定义的系统调用函数指针表数组 `sys_call_table[]` 中项的索引值（也就是数组的下标）。

在 Linux 中规定 `int 0x80` 指令是系统调用的总入口，系统调用号存放在 EAX 寄存器中，应用程序通过系统调用传递给内核函数的参数依次存放于 EBX、ECX 和 EDI 这三个寄存器中（即系统调用最多只能有 3 个参数）。

Linux 系统调用的执行过程为：首先，应用程序准备好系统调用需要的参数，并直接或间接（通过库函数）发出一个调用请求，即调用 `int 0x80` 指令；然后，该指令通过陷阱处理机制（中断处理机制的一种），使该系统调用进入内核的入口点 `_system_call` 函数（`kernel/system_call.s` 文件中第 101 行）；最后，由 `_system_call` 函数找到系统调用对应的内核函数，该内核函数将被执行并通过 EAX 寄存器返回结果。

详细内容请读者阅读《Linux 内核完全注释》第 5 章第 5 节和第 8 章第 5 节的内容。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。

3.2 添加新的系统调用

为 Linux 0.11 添加一个新的系统调用 `max` 函数，该函数实现比较两个参数的大小并将较大值返回的功能。步骤如下：

1. 首先要为新系统调用分配一个唯一的系统调用号，以原来的最大系统调用号为基础加 1 作为新的系统调用号。在“项目管理器”窗口中双击打开 `include/unistd.h` 文件，在第 162 行添加新的系统调用号，如图 4-1：

```

161| #define __NR_uselib 86
162| #define __NR_max 87
163| // 以下定义系统调用嵌入式汇编宏函数。

```

图 4-1 添加新的系统调用号

2. 添加新系统调用号的同时，也要使系统调用总数在原来的基础上增加 1。打开 kernel/system_call.s 文件，修改在第 73 行定义的系统调用总数，如下图：

```

71| sa_restorer = 12    # 返回恢复执行的地址位置。参见kern
72|
73| nr_system_calls = 88    # Linux 0.11 版内核中的系统调用
74|
75| /*

```

图 4-2 修改 Linux 内核的系统调用总数

3. 在 include/linux/sys.h 文件中的第 88 行使用 C 语言声明内核函数的原型，如图 4-3 所示。注意，这里定义的函数原型并不需要与函数的定义完全一致，只需要定义函数的返回值为 int 类型，参数为空，能够让 sys_max 标识符代表一个函数名称即可。

```

86| extern int sys_readlink();
87| extern int sys_uselib();
88| extern int sys_max();
89| // 系统调用函数指针表。用于系统调用中断处理程序(int

```

图 4-3 使用 C 语言声明内核函数的原型

在此文件的最后，向系统调用函数指针表 sys_call_table[] 中添加新系统调用函数的指针（注意，系统调用号必须与系统调用内核函数指针在系统调用函数表中的索引一一对应），如图 4-4 所示：

```

176| sys_readlink,    //85
177| sys_uselib,      //86
178| sys_max          //87
179| };
180|

```

图 4-4 在系统调用函数指针表中增加内核函数的指针

4. 在 kernel/sys.c 文件的最后编写代码，实现新系统调用对应的内核函数，如图 4-5：

```

353|
354| int sys_max( int max1, int max2 )
355| {
356|     return max1>max2?max1:max2;
357| }
358|

```

图 4-5 系统调用对应的内核函数

5. 按 F7 生成 Linux 0.11 内核，修改语法错误直到生成成功。接下来需要在 Linux 0.11 中编写一个应用程序来测试新系统调用是否添加成功。

3.3 在应用程序中测试新系统调用

1. 按 F5 启动调试。
2. 待 Linux 0.11 启动后，使用 vi 编辑器新建一个 main.c 文件。编辑 main.c 文件中的源代码（如图 4-6 所示）。其中，需要定义 __LIBRARY__ 宏及包含 unistd.h 头文件，还需要再次定义 __NR_max 宏，并使用 _syscall2 宏（在文件

include/unistd.h 中的第 197 行定义)对系统调用函数进行定义, 当该宏展开时, 就会在源代码文件中使用 C 语言添加 max 函数的完整实现。

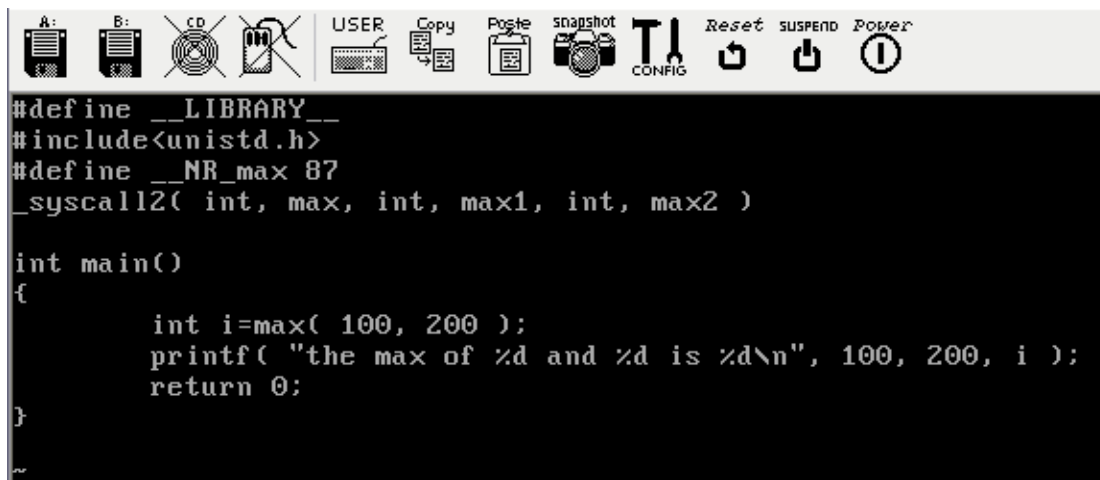


图 4-6 编辑 main.c 文件

3. 使用命令 `gcc main.c -o main` 生成可执行文件 main。
4. 执行 `sync` 命令, 将文件保存到磁盘。
5. 执行 `chmod +x main` 命令为 main 文件添加可执行权限。
6. 运行 main。如果 main 能输出正确的结果, 则说明新系统调用添加成功。

可以按照下面的步骤查看 _syscall12 宏展开后得到的 max 函数:

1. 依次执行下面的命令。其中 gcc 的 -E 选项是将源文件进行预处理, 它会将源文件中的头文件以及宏都展开。

```
gcc -E main.c -o main.txt
sync
mcopy main.txt b:main.txt
```
2. 结束调试后, 在“项目管理器”窗口中双击 floppyb.img 文件, 使用软盘编辑器工具打开。将其中的 main.txt 文件复制到 windows 的某个文件夹中。
3. 将文件夹中的 main.txt 文件拖动到 Linux lab 中释放, 在文件的最后部分就可以看到 _syscall12 宏展开后得到的 max 函数了。可以重点学习一下通过在 C 代码中嵌入 AT&T 汇编, 调用 int 0x80 号中断, 并使用寄存器传递参数和返回值的过程。

3.4 调试系统调用执行的过程

根据 _syscall10、_syscall11、_syscall12、_syscall13 这四个宏(在文件 include/unistd.h 中的第 167 行定义)可知系统调用执行的过程如下: 应用程序通过调用 int 0x80 中断, 进入内核中的系统调用总入口 _system_call 函数, 该函数以存放在 EAX 寄存器中的系统调用号作为系统调用函数指针表的索引, 找到相应的内核函数, 并通过 EBX、ECX、EDX 将参数传递给内核函数, 最后通过 call 指令执行该内核函数并使用 EAX 寄存器返回结果。

为了调试系统调用执行的过程, 需要在系统调用总入口 _system_call 函数中添加一个断点。但是, 因为 Linux 操作系统在启动的过程中会多次产生 int 0x80 调用, 在 _system_call 处的断点就会被多次命中。为了减少断点的命中次数, 需要将此断点设置为一个条件断点。调试的步骤如下:

1. 选择“调试”菜单中的“停止调试”, 结束之前的调试过程。
2. 打开 kernel/system_call.s 文件, 在标号 _system_call 后的第一行汇编代码处(第

- 102 行) 添加一个断点。
3. 选择“调试”菜单“窗口”中的“断点”，激活断点窗口。
4. 在“断点”窗口中右键点击刚刚添加的断点，在弹出的菜单中选择“条件”，弹出断点条件对话框，在编辑框中设置断点条件 `$eax==87` 后按“确定”，如下图：

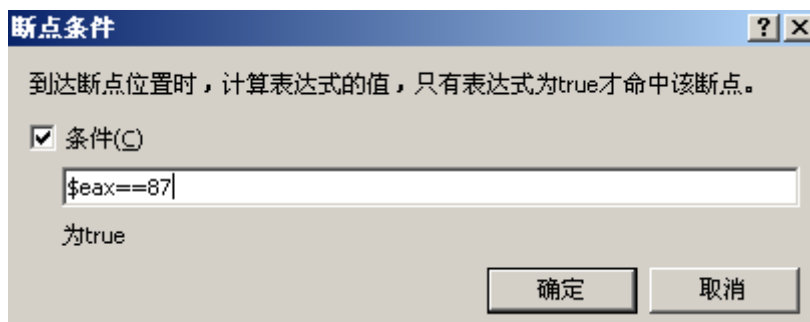


图 4-7 设置断点条件

5. 按 F5 启动调试，待 Linux 0.11 启动后运行 main，会命中刚刚添加的条件断点。
6. 选择“调试”菜单“窗口”中的“监视”，激活“监视”窗口。
7. 在监视窗口的“名称”列中输入“\$eax”后，按回车，会发现 EAX 寄存器中的值为 0x57 (十进制为 87，也可以点击工具栏上的“十六进制”按钮切换为十进制显示)，和 max 系统调用的调用号一致，说明应用程序正在调用 max 系统调用函数。

监视		
名称	值	类型
\$eax	0x57	int

图 4-8 使用“监视”窗口查看 EAX 寄存器的值

在“监视”窗口查看 EBX 和 ECX 寄存器的值，存放的分别是 max 函数的参数 100 和 200。

监视		
名称	值	类型
\$eax	87	int
\$ebx	100	int
\$ecx	200	int

图 4-9 使用“监视”窗口查看 EBX 和 ECX 寄存器的值

8. 按 F10 单步调试，直到黄色箭头指向第 110 行。其中第 103 行进行错误检查，第 104-106 行将各个段寄存器的值压入堆栈进行现场保护，第 107-109 行将保存在 EBX、ECX 和 EDX 寄存器中的参数压入堆栈，这与 C 语言参数从右到左压入堆栈的顺序是一致的。
9. 按 F10 继续单步调试，直到黄色箭头指向第 119 行。其中，第 119 行使用 EAX 寄存器存放的系统调用号作为系统调用函数指针表的下标，调用对应的系统调用内核函数。
10. 按 F11 调试进入 kernel/sys.c 文件中的系统调用对应的内核函数，如下图：

```

353
354 □ int sys_max( int max1, int max2 )
355 {
356     return max1>max2?max1:max2;
357 }
358

```

图 4-10 进入系统调用对应的内核函数

11. 按 F10 单步调试，直到从内核函数返回到 kernel/system_call.s 文件中的第 120 行。从内核函数返回后，返回值会存放在 EAX 寄存器中，如下图：

监视		
名称	值	类型
\$eax	200	int
\$ebx	100	int
\$ecx	200	int

图 4-11 EAX 寄存器存放返回值

12. 按 F10 单步调试，直到黄色箭头指向第 168 行。其中，因为后续工作（进程调度、信号处理）会用到 EAX 寄存器，所以 120-121 行会将 EAX 存放的返回值（200）入栈，并把当前任务数据结构地址存入 EAX 寄存器，122-160 行会进行进程调度以及信号的处理工作，161 行-167 行，恢复现场，恢复通用寄存器以及段寄存器，与保护现场时的顺序相反。此时查看监视窗口中的 EAX，EBX 和 ECX 寄存器的值，分别恢复为 200（返回值），100（参数一），200（参数二），如下图：

监视		
名称	值	类型
\$eax	200	int
\$ebx	100	int
\$ecx	200	int

图 4-12 出栈操作恢复了 EAX，EBX 和 ECX 寄存器的值

13. 按 F5 继续运行，第 168 行的 iret 指令从 0x80 中断返回到 main 程序中。

四、思考与练习

- 参考《Linux 内核完全注释》第 8.5.3.3 节的内容，编写一个汇编程序，直接使用系统调用。
- 在 Linux 0.11 内核中添加两个系统调用函数 Iam 和 Whoami，函数原型如下：
 - int Iam(const char* name); 将字符串 name 的内容保存到内核中，返回值是拷贝的字符数，如果 name 长度大于 32，则返回-1，并置 errno 为 EINVAL。
 - int Whoami(char* name, int size); 将 Iam 保存到内核中的字符串拷贝到数据缓冲区 name 中，size 为数据缓冲区 name 的长度，返回值是拷贝的字符数。如果 size 小于所需空间，则返回-1，并置 errno 为 EINVAL。

编写两个应用程序。其中，Iam 应用程序调用 Iam 函数，并使用命令行中的第一个参数作为 Iam 函数的参数；Whoami 应用程序调用 Whoami 函数，并打印输出获取的字符串。测试效果如下图：

```
[/usr/root]# Iam 123
[/usr/root]# Whoami
123
[/usr/root]#
```

图 4-13 测试应用程序的执行结果

提示：

1. `errno` 是一个传统的错误代码返回机制。当一个函数调用出错时，会把错误值存放到全局变量 `errno` 中，调用者就可以通过判断 `errno` 来决定如何应对错误，请读者使用“查找和替换”对话框找到 `errno` 是在哪里定义的。
2. 要在内核中保存字符串数据，需要在内核中定义一个字符数组全局变量。可以把该字符数组和两个系统调用的内核函数定义在 `kernel/sys.c` 文件中。
3. `Iam` 和 `Whoami` 两个系统调用使用了指针参数传递用户地址空间的逻辑地址（字符串的开始地址），若在内核空间直接访问这个地址，访问的仍然是内核空间的数据，而不是用户空间的数据。所以，在 `Iam` 函数中需要在一个循环中重复调用 `get_fs_byte` 函数，获取用户空间中的数据；在 `Whoami` 函数中需要在一个循环中重复调用 `put_fs_byte` 函数，将内核空间中的数据复制到用户空间（详情请参考 `include/asm/segment.h` 文件）。

实验五 进程的创建

实验性质：验证

建议学时：2 学时

实验难度：★★★★☆☆

一、实验目的

- 掌握创建子进程和加载执行新程序的方法，理解创建子进程和加载执行程序的不同。
- 调试跟踪 fork 和 execve 系统调用函数的执行过程。

二、预备知识

程序通常是指一个可执行文件，而进程则是一个正在执行的程序实例。利用分时技术，在 Linux 操作系统上可以同时运行多个进程。分时技术的基本原理是把 CPU 的运行时间分成一个个规定长度的时间片，让每个进程在一个时间片内运行。当一个进程的时间片用完时，操作系统就利用调度程序切换到另一个进程去运行。因此，本质上对于具有单个 CPU 的机器来说，某一时刻只有一个进程在运行，但是由于进程运行的时间片比较短（通常为几十毫秒），所以给用户的感觉是多个进程在同时运行。

对于 Linux 0.11 内核来讲，系统最多可有 64 个进程同时存在。除了第一个进程用“手工”建立以外，其余的都是现有进程使用系统调用 fork 函数创建的新进程。被创建的进程称为子进程，创建者则称为父进程。Linux 内核使用进程标识号来标识每个进程。进程由可执行的指令代码、数据区和堆栈区组成。进程中的指令代码和数据区分别对应于可执行文件中的代码段和数据段，而堆栈区是在创建进程时由操作系统为其分配的。每个进程只能执行自己的代码和访问自己的数据区和堆栈区。

Linux 操作系统内核通过进程表（数组）对进程进行管理，每个进程在进程表中占有一项。这个进程表数组在 kernel/sched.c 文件的第 79 行定义，数组长度是 64。进程表项是一个 task_struct 任务数据结构的指针。任务数据结构（也称为进程控制块 PCB 或进程描述符 PD）定义在 include/linux/sched.h 文件中的第 136 行。其中保存着用于控制和管理进程的所有信息，主要包括进程当前运行的状态、信号、进程号、父进程号、运行时间累计值、正在使用的文件和本任务的局部描述符以及任务状态段等。

详细内容请读者阅读《Linux 内核完全注释》第 5 章第 7 节的内容。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 使用“Linux011 应用程序”模板新建一个 Linux011 应用程序项目。

3.2 调用 fork 函数创建子进程

编写一个 Linux011 应用程序程序，在程序中调用 fork 函数创建子进程，并分析程序运行的结果。步骤如下：

1. 编辑 LinuxApp.c 文件，将 main 函数修改为如下的代码：

```
int main( int argc, char * argv[] )
{
    int pid;
```

```

    printf( "%d hello world\n", getpid() );
    pid = fork();
    if( pid != 0 )
    {
        printf( "%d parent process\n", getpid() );
    }
    else
    {
        printf( "%d child process\n", getpid() );
    }
    printf( "%d exit\n", getpid() );
    return 0;
}

```

其中的 `getpid` 函数是一个系统调用函数，返回当前进程的进程号。

2. 按 F7 生成项目。
3. 按 F5 启动调试
4. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：

```
mcopy b:linuxapp.exe app
```

5. 给 `app` 文件添加可执行权限，命令如下：

```
chmod +x app
```

6. 执行 “`sync`” 命令，将对文件的更改保存到磁盘。

7. 查看可执行文件 `app` 的信息，

```
ls -l app
```

将 `app` 文件的大小记录下来，在后面调试跟踪 `fork` 执行过程时会用到此值。

8. 运行可执行文件 `app`，分析运行结果。

`fork` 是一个系统调用函数。该系统调用函数在执行时，会在进程表中创建一个与调用此函数的进程（父进程）几乎完全一样的新的进程表项（子进程），子进程与并父进程执行同样的代码，但该子进程拥有自己的数据空间和环境参数。在 `fork` 函数的返回位置处，父进程将恢复执行，而子进程也从相同的位置开始执行。在父进程中，调用 `fork` 返回的值是子进程的进程标识号 PID，而在子进程中 `fork` 函数返回的值是 0（这正是 `fork` 的神奇之处，调用一次，返回两次）。

在父进程中可以调用 `wait` 函数或 `waitpid` 函数阻塞父进程，直到子进程退出。这两个函数的原型在 `include/sys/wait.h` 文件中定义如下：

```

pid_t wait( pid_t * wait_loc )
pid_t waitpid( pid_t pid, pid_t * wait_loc, int options )

```

按照下面的步骤继续修改程序：

1. 在 “`printf(“%d parent process\n”, getpid());`” 语句前，添加一行语句：

```
wait( NULL );
```
2. 生成项目，启动调试，从 B 盘复制应用程序后执行，观察结果与之前有何不同。
3. 练习使用 `waitpid` 函数替换 `wait` 函数。

3.3 调试跟踪 fork 函数的执行过程

为了调试跟踪 fork 函数的执行过程，需要在系统内核中添加条件断点，步骤如下：

1. 再启动一个 Linux Lab 程序，使用“Linux011 Kernel”模板新建一个 Linux011 Kernel 项目。
2. 用之前创建的 Linux011 应用程序项目的硬盘镜像文件 harddisk.img 覆盖当前 Linux011 Kernel 项目的 harddisk.img 文件，这样就可以使用之前生成的 app 文件了。
3. 打开 kernel/system_call.s 文件，在第 102 行添加一个条件断点，条件为：
`$eax==2 && current->executable->i_size==文件大小`
“文件大小”就是之前记录的应用程序可执行文件 app 的大小，这样就只有在执行 app 中的 fork 函数时，才会命中此条件断点。“\$eax==2”中的 2 是 fork 函数的系统调用号。current 是一个全局变量（在 kernel/sched.c 文件的第 76 行定义），总是指向当前正在运行的进程的控制块。
4. 按 F5 启动调试，运行 app 应用程序，命中断点。

此时，由于在应用程序中调用了 fork 函数，所以就进入了 int 0x80 的中断处理程序并命中了断点。接下来会调用 fork 系统调用的内核函数，继续按照下面的步骤调试：

1. 在“监视”窗口监视 last_pid 的值和 current->pid 的值。全局变量 last_pid（在文件 kernel/fork.c 的第 30 行定义）记录了最新的进程号。current->pid 的值是当前正在运行的进程的进程号。
2. 选择菜单“调试”中的“快速监视”，打开“快速监视”对话框。在对话框的“表达式”编辑控件中输入“*current”就可以查看当前进程的控制块中的所有值。其中“state=0”表示当前进程（即 app 程序创建的进程）正处于运行态；“counter=11”表示其剩余时间片的大小；“priority=15”表示其优先级；“father=4”表示其父进程的进程号。
3. 关闭“快速监视”对话框后，按 F10 单步调试至第 119 行，再按 F11 进入 fork 系统调用的内核函数，可以看到其内核函数是一个汇编函数。
4. 按 F10 单步调试至第 272 行。此时，第 271 行的 find_empty_process 函数（在文件 kernel/fork.c 的第 169 行定义）已经执行完毕，为新进程取得了一个不重复的进程号，并找到了一个未被使用的任务数组项返回其索引（在 EAX 寄存器中返回）。查看“监视”窗口，可看到 last_pid 的值已经发生了变化，该值即为子进程的进程号。
5. 按 F10 单步调试至第 279 行，然后按 F11 进入 copy_process 函数。在第 278 行将 EAX 寄存器的值最有一个压入堆栈，也就意味着 copy_process 函数的第一个参数为子进程在任务数组中的索引。

copy_process 函数（在文件 kernel/fork.c 的第 89 行定义）是 fork 过程中调用的一个重要函数，该函数为子进程申请了一个进程控制块，并完成初始化工作。按照下面的步骤调试此函数：

1. 按 F10 单步执行至第 103 行。其中，第 98 行为子进程的控制块分配了内存；第 101 行将新创建的子进程控制块的指针放入了任务数组中，数组索引由第一个参数指定。此时在“快速监视”窗口查看表达式 *p 的值，可以看到新创建的子进程控制块中各个成员的值都为 0。

2. 按 F10 单步执行至第 104 行。第 103 行的代码非常关键，此行代码将 `current` 指向的父进程控制块中的内容完全复制到了 `p` 指向的子进程控制块中，也就是子进程完全继承了父进程的各种资源。此时在“快速监视”窗口中分别查看表达式 `*p` 和 `*current` 的值，就可以发现它们的值是完全相同的。这就可以解释很多现象，例如子进程和父进程的优先级相同，而且由于两个进程控制块中保存的 EIP 寄存器的值相同，所以，子进程和父进程会从相同的位置（fork 函数返回的位置）继续运行。
3. 在第 139 行添加一个断点，按 F5 继续调试，命中后删除该断点。由于子进程控制块除了从父进程控制块继承资源之外，还需要设置自己特有的资源，所以，第 104 行设置子进程为“不可中断等待状态”；第 105 行设置子进程的进程号；第 106 行设置子进程的父进程号；第 122 行将 `p->tss.eax` 的值设置为 0，就是 fork 函数在子进程中返回 0 的原因，随后设置子进程控制块中的其他成员。
4. 在第 165 行添加一个断点，按 F5 继续调试，命中后删除该断点。刚刚执行的这段代码对父进程控制块及子进程控制块又做了一些调整。当子进程控制块初始化完毕后，在第 163 行将子进程控制块中的状态标志置为“就绪状态”。此时，在“快速监视”窗口分别查看 `*current` 和 `*p` 的值，比较二者的异同。
5. 按 F10 单步调试，直到从 `copy_process` 函数返回到 `kernel/system_call.s` 文件中的第 280 行。`copy_process` 函数的返回值是子进程的进程号，会被放入 EAX 寄存器中，也就是父进程从 fork 函数返回时得到的返回值。
6. 按 F10 单步调试，直到从汇编函数返回到 `kernel/system_call.s` 文件中的第 120 行。
7. 继续按 F10 单步调试，直到第 133 行。可以看到在从 fork 系统调用返回之前，并没有执行进程调度 `reschedule` 函数，所以父进程会继续运行。但是，如果从 fork 返回后，在父进程调用了 `wait` 或 `waitpid` 系统调用函数等待子进程结束的情况下，就会在从这些系统调用函数返回之前执行进程调度 `reschedule` 函数，从而让处于就绪状态的子进程先运行，待子进程退出后父进程再运行。
8. 按 F5 继续调试，可执行文件运行结束。

3.4 调用 `execve` 函数加载执行一个新程序

使用 fork 系统调用函数可以为父进程创建一个子进程，但是子进程和父进程执行的是同一个程序。为了让父进程加载执行一个新程序，可以使用 `execve` 系统调用函数。首先按照下面的步骤编写一个 Linux 0.11 的应用程序，作为将要被 `execve` 函数加载的程序：

1. 新建一个 Linux011 应用程序项目。
2. 编辑 `LinuxApp.c` 文件中的 `main` 函数，代码如下：

```
int main( int argc, char * argv[] )
{
    printf("%d hello world\n", getpid() );
    return 0;
}
```
3. 按 F7 生成项目，F5 启动调试。
4. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：

```
mcopy b:linuxapp.exe newapp
```
5. 为 `newapp` 文件添加可执行权限，命令如下：

```
chmod +x newapp
```

6. 执行“sync”命令后运行 newapp，观察运行结果。
7. 结束本次调试。

接下来编写调用 execve 函数的应用程序：

1. 再次编辑 main 函数，代码如下：

```
int main( int argc, char * argv[] )
{
    printf("%d hello execve\n", getpid() );
    execve("newapp", NULL, NULL );
    printf("%d hello exit\n", getpid() );
    return 0;
}
```

int execve(char* file, char** argv, char** envp) 函数用来加载执行一个新程序。参数 file 是需要被加载的程序文件名，参数 argv 是传递给新程序的命令行参数指针数组，参数 envp 是传递给新程序的环境变量指针数组。

2. 按 F7 生成项目，F5 启动调试；
3. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：

```
mcopy b:linuxapp.exe app
```

4. 给 app 添加可执行权限，命令如下：

```
chmod +x app
```

5. 查看可执行文件 app 的详细信息，命令如下：

```
ls -l app
```

将文件的大小记录下来，在后面调试跟踪 execve 函数的执行过程时会用到此值。

6. 执行“sync”命令后，运行可执行文件 app，观察运行结果。
7. 结束本次调试。

execve 系统调用会清理掉当前进程的页目录和页表项，并释放对应的物理页，然后为新加载的可执行文件中的指令和数据重新申请内存，并配置到当前进程的控制块中，还会重新设置代码执行的起始位置。此时当前进程的代码和数据将完全被新程序替换掉，并在该进程中开始执行新程序的代码。Linux 0.11 为加载执行新程序提供了 exec 函数族，exec 函数族的相关函数一般在库函数中实现，但是最终都要调用 execve 这个系统调用。

3.5 调试跟踪 execve 函数的执行过程

为了调试跟踪 execve 函数的执行过程，需要在系统内核中添加条件断点，步骤如下：

1. 再启动一个 Linux Lab 程序，使用“Linux011 Kernel”模板新建一个 Linux011 Kernel 项目。
2. 用之前创建的 Linux011 应用程序项目的硬盘镜像文件 harddisk.img 覆盖当前 Linux011 Kernel 项目的 harddisk.img 文件，这样就可以使用之前生成的 app 和 newapp 文件了。
3. 打开 kernel/system_call.s 文件，在第 102 行添加一个条件断点，条件为：
\$eax==11 && current->executable->i_size==文件大小
“\$eax==11”中 11 是 execve 函数的系统调用号。“文件大小”是之前记录的应用程序可执行文件 app 的大小。
4. 按 F5 启动调试，运行 app 应用程序，命中断点。

此时，由于在应用程序中调用了 `execve` 函数，所以就进入了 `int 0x80` 的中断处理程序并命中了断点。接下来会调用 `execve` 系统调用的内核函数，继续按照下面的步骤调试：

1. 在“监视”窗口中查看 `last_pid` 的值，在后续的调试过程中注意此值是否发生变化。
2. 按 F10 单步调试到第 119 行，按 F11 进入到 `execve` 系统调用对应的汇编函数 `sys_execve`，黄色箭头指向第 260 行。
3. 按 F10 单步调试到底 262 行，按 F11 进入到 `do_execve` 函数中，该函数完成加载执行新程序的主要功能。
4. 在第 314 行添加一个断点，按 F5 继续调试，命中后删除该断点。第 303 到第 304 行初始化参数和环境变量空间的页面指针数组。第 306 行取得可执行文件对应的 `i` 节点号。第 309 到第 310 行计算参数个数和环境变量个数。
5. 在第 472 行添加一个断点，按 F5 继续调试，命中后删除该断点。该过程主要完成对文件合法性的检查以及参数和环境变量的复制。
6. 在第 494 行添加一个断点，按 F5 继续调试，命中后删除该断点。第 472 到第 474 行释放进程原始的可执行文件的 `i` 节点，并使其指向新程序的可执行文件的 `i` 节点，接下来是对进程控制块信号句柄和协处理器的处理。
7. 按 F10 单步调试到第 508 行。第 494 到第 496 行创建参数和环境变量指针表，并返回该堆栈指针。第 499 行设置代码段、数据段以及堆栈段信息。第 503 到第 509 行设置进程堆栈开始字段所在页面以及用户 ID 和组 ID。
8. 在第 515 行添加一个断点，按 F5 继续调试，命中后删除该断点。第 508 到第 509 行初始化 `bss` 段数据。第 513 到第 514 行将堆栈上的代码指针替换为新程序的入口点地址，并将堆栈指针替换为新程序的堆栈指针。
9. 按 F10 单步调试，`do_execve` 函数返回到 `sys_execve` 函数。
10. 按 F5 继续调试，app 运行结束。

体会 `execve` 函数的执行过程，该过程中并没有申请新的进程控制块，同时 `last_pid` 的值也没有发生变化，只是对当前进程的控制块进行了相应的修改，从而加载执行了另一个程序。

3.6 fork 与 execve 的区别与联系

`fork` 会为子进程重新申请一个进程控制块 (`task_struct`)，并拷贝父进程的进程控制块信息到子进程的进程控制块中，再对子进程的控制块做简单的修改，使子进程与父进程执行同样的程序。`execve` 并没有申请新的进程控制块，而是直接修改当前进程的进程控制块，并开始执行一个新程序。

在为 Linux 开发应用程序时，往往会同时使用 `fork` 和 `execve` 函数。一个程序在使用 `fork` 函数创建了一个子进程时，通常会在该子进程中调用 `execve` 函数加载执行另一个新程序，例如：

```
if( fork() )
{
    /* parent process */
}
else
{

```

```
/* child process */  
execve(...);  
}
```

四、思考与练习

1. 模仿 3.2 中的源代码，在 Linux 应用程序中使用 for 语句编写一个循环，使父进程能够循环创建 10 个子进程，每个子进程在输出自己的 pid 后退出，父进程等待所有子进程结束后再退出。
2. 结合 3.2、3.4 和 3.6 中的内容编写一个 Linux 应用程序，在 main 函数中使用 fork 函数创建一个子进程，在子进程中使用 execve 函数加载执行另外一个程序的可执行文件（例如 newapp），并且让父进程在子进程退出后再结束运行。
3. 读者如果在 Linux 0.11 刚刚启动后按 F1 键，可以看到当前系统中存在 4 个进程，进程号依次为 0、1、4、3。请读者结合 Linux 0.11 的初始化，fork 函数的执行过程以及 Linux 0.11 使用进程表（数组）管理进程的方式，说明为什么进程号的顺序不是 0、1、3、4，而是 0、1、4、3。

实验六 进程的状态和进程调度

实验性质：验证、设计

建议学时：2 学时

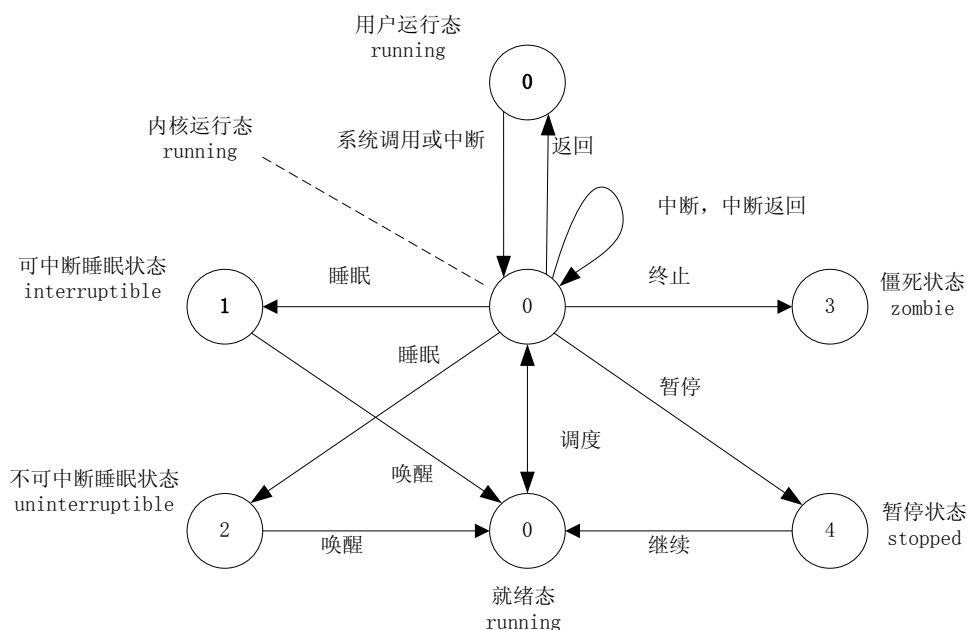
实验难度：★★★★☆

一、实验目的

- 调试进程在各种状态间的转换过程，熟悉进程的状态和转换。
- 通过对进程运行轨迹的跟踪来形象化进程的状态和调度。
- 掌握 Linux 下的多进程编程技术。

二、预备知识

一个进程在其整个生命周期内，不同阶段可能具有不同的进程状态（在 `include/linux/sched.h` 文件的第 20 行定义）。一个进程的状态由其进程控制块中的 `state` 字段指定。



- **运行状态 (TASK_RUNNING)**。当进程正在被 CPU 执行时，或已经准备就绪可由调度程序调度时，则称该进程处于运行状态。若此时没有被 CPU 执行，则称其处于**就绪运行状态**。进程可以在内核态运行，也可以在用户态运行。当一个进程在内核代码中运行时，称其处于内核运行态，或简称内核态；当一个进程正在执行用户自己的代码时，称其处于用户运行态，或简称用户态。当进程所需的系统资源已经可用时，进程就会被唤醒，从而进入就绪运行状态。这些状态在内核中表示方法相同，都被称为处于 TASK_RUNNING 状态。例如，当一个新进程刚刚被创建后就处于就绪运行态。

- **可中断睡眠状态 (TASK_INTERRUPTIBLE)**。当进程处于可中断睡眠状态时，系统不会调度该进程执行。当系统产生一个中断或者释放了该进程正在等待的资源，或者该进程收到一个信号时，该进程都会被唤醒，从而转换到就绪运行态。
- **不可中断睡眠状态 (TASK_UNINTERRUPTIBLE)**。除了不会因为收到信号而被唤醒外，该状态与可中断睡眠状态类似。但处于该状态的进程只有被 `wake_up` 函数明确唤醒时才能转换到就绪运行状态。该状态常在进程需要不受干扰地等待或者所等待事件很快就会发生时使用。
- **暂停状态 (TASK_STOPPED)**。当进程收到信号 `SIGSTOP`、`SIGTSTP`、`SIGTTIN` 或 `SIGTTOU` 时就会进入暂停状态。可向处于暂停状态的进程发送 `SIGCONT` 信号，让其转换到就绪运行状态。进程在调试期间接收到任何信号均会进入该状态。在 Linux 0.11 中，还未实现对该状态的转换处理，处于该状态的进程将被作为进程终止来处理。
- **僵死状态 (TASK_ZOMBIE)**。当子进程已停止运行，但其父进程还没有调用 `wait` 函数询问其状态时，则称该进程处于僵死状态。因为，父进程需要获取子进程停止运行的信息（例如获取子进程的退出码），所以，此时子进程的进程控制块还需要保留。一旦父进程通过调用 `wait` 函数取得了子进程的信息，则处于该状态的子进程的进程控制块就会被释放掉。

当一个进程的时间片用完时，操作系统会使用调度程序强制切换到其他的进程去执行。另外，如果进程在内核态执行时需要等待系统的某个资源，此时该进程就会调用 `sleep_on` 或 `interruptible_sleep_on` 函数自愿地放弃 CPU 的使用权，从而让调度程序去执行其他进程，此进程则进入睡眠状态（`TASK_UNINTERRUPTIBLE` 或 `TASK_INTERRUPTIBLE`）。只有当进程从“内核运行态”转换到“睡眠状态”时，内核才会进行进程切换操作。在内核态下运行的进程不能被其他进程抢占，而且一个进程不能改变另一个进程的状态。为了避免进程切换时造成内核数据错误，内核在执行临界区代码时会禁止一切中断。

内核中的调度程序 `schedule`（在文件 `kernel/sched.c` 中的第 135 行定义）用于选择下一个要运行的进程。这种选择运行机制是多任务操作系统的基础。调度程序可以被看作一个在所有处于运行状态的进程之间分配 CPU 运行时间的管理器。为了能有效地使用系统资源，又能使各个进程有较快的响应时间，就需采用一定的调度策略，在 Linux 0.11 中采用了基于优先级排队的调度策略。

详细内容请读者阅读《Linux 内核完全注释》第 5 章第 7 节的内容。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 应用程序项目。

3.2 编写一个多进程程序

请读者按照下面的步骤使用 `fork` 函数编写一个多进程程序，一方面学习多进程编程技术，另一方面，通过观察多个进程的运行过程，先从现象上对进程的调度过程有一个初步认识。

1. 编辑 `LinuxApp.c` 文件中的 `main` 函数，让父进程新建三个子进程，并分别输出子进程 `id` 以及其父进程 `id`。代码如下：

```
int main( int argc, char * argv[] )
{
```

```

    if( 0 == fork() )
    {
        printf("child process pid=%d ppid=%d line=%d\n",
            getpid(), getppid(), __LINE__);
    }
    else if( 0 == fork() )
    {
        printf("child process pid=%d ppid=%d line=%d\n",
            getpid(), getppid(), __LINE__);
    }
    else if( 0 == fork() )
    {
        printf("child process pid=%d ppid=%d line=%d\n",
            getpid(), getppid(), __LINE__);
    }
    else
    {
        wait( NULL );
        printf("parent process pid=%d ppid=%d\n",getpid(), getppid());
    }
    return 0;
}

```

getppid 函数可以在子进程中获取父进程的 id。“__LINE__”是 GCC 编译器提供的一个预定义宏，它的值是其所在的行号。

2. 按 F7 生成项目，按 F5 启动调试。
3. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：
`mcopy b:linuxapp.exe app`
4. 为 app 文件添加可执行权限，命令如下：
`chmod +x app`
5. 执行“sync”命令。
6. 执行可执行文件 app，观察各个进程执行的顺序并说明原因，理解进程在生命周期中状态的转换过程和进程调度过程。
7. 结束此次调试。

3.3 跟踪进程的运行轨迹

跟踪进程的运行轨迹，本质上就是跟踪并记录进程在其生命周期中的状态转换过程和调度过程。进程在生命周期中状态的转换是由操作系统的调度程序实现的。所以要实现对进程运行轨迹的跟踪，不仅要进程在其生命周期的各个状态有一个全面的了解，而且还要对进程的调度有一个透彻的理解。

在开始记录进程的运行轨迹之前，需要先在 Linux 0.11 内核中添加一个写日志文件的功能，然后使用此功能将进程状态转换的日志信息写入一个文本文件。

3.3.1 在系统初始化时打开日志文件 process.log

为了记录操作系统从启动到关机过程中所有进程的运行轨迹,需要在硬盘上维护一个日志文件/var/process.log。为了能尽早开始记录日志,应当在内核初始化时就打开 process.log 文件。

首先使用 Linux011 Kernel 模板新建一个项目,按照下面的内容修改其源代码:

内核的入口函数是 init/main.c 文件中的 start 函数,其中从第 181 行开始的一段代码是:

```
.....
move_to_user_mode();
if( !fork() ) {          /*we count on this going ok*/
    init();
}
```

这段代码是在进程 0 中运行的,先切换到用户模式,然后第一次调用 fork 函数创建子进程 1,子进程 1 接着调用了本文件中的 init 函数继续进行初始化工作。

在 init 函数中从第 227 行开始的一段代码是:

```
.....
setup( ( void * )&drive_info );
( void ) open("dev/tty0", O_RDWR, 0 );
( void ) dup( 0 );
( void ) dup( 0 );
.....
```

这段代码建立了文件描述符 0、1 和 2,它们分别是 stdin、stdout 和 stderr,即标准输入、标准输出和标准错误。可以在这里把 process.log 文件关联到文件描述符 3。

为了能尽早访问 process.log 文件,可以让上述建立文件描述符的工作在进程 0 中完成。所以,将这一段代码从 init 函数中移动(是剪切不是复制!)到 start 函数中,而且要放在调用 move_to_user_mode 之后(不能再靠后了!),同时加上打开 process.log 文件的代码。修改后的 start 函数如下:

```
.....
    move_to_user_mode();
    /***** your code begin *****/
    setup( ( void * )&drive_info );
    ( void ) open("dev/tty0", O_RDWR, 0 );
    ( void ) dup( 0 );
    ( void ) dup( 0 );
    ( void ) open("/var/process.log", O_CREAT | O_TRUNC | O_WRONLY, 0666 );
    /***** your code end *****/
    if( !fork() ) {          /*we count on this going ok*/
        init();
    }
.....
```

调用 open 函数打开 process.log 文件。第二个参数的含义是建立只写文件,并且如果文件已存在则清空已有内容。第三个参数设置文件权限为所有人可读可写。

这样，文件描述符 0、1、2 和 3 就在进程 0 中建立了，根据 fork 函数的原理，进程 1 就会从进程 0 中继承这些文件描述符，而且，由于此后所有的进程都是进程 1 的子进程，当然也就会继承这些文件描述符了。

3.3.2 编写 fprintfk 函数用于向 process.log 文件写入数据

process.log 文件将会被用来记录进程状态切换的轨迹，但是，所有进程的状态切换工作都是在内核状态下进行的，此时 write 系统调用函数失效（其原理等同于不能在内核状态下调用 printf 函数，而只能调用 printk 函数）。编写可以在内核状态下被调用的 fprintfk 函数的难度较大，所以这里直接给出源代码，主要是参考了 printk 函数和 sys_write 函数而写成的：

```
#include<linux/sched.h>
#include<sys/stat.h>

static char logbuf[1024];

int fprintfk( int fd, const char * fmt, ... )
{
    va_list args;
    int i;
    struct file * file;
    struct m_inode * inode;
    va_start (args, fmt);
    i = vsprintf (logbuf, fmt, args);
    va_end (args);

    if( fd<3 )
    {
        __asm__ ("push %%fs\n\t"
                "push %%ds\n\t"
                "pop %%fs\n\t"
                "pushl %0\n\t"
                "pushl $_logbuf\n\t"
                "pushl %1\n\t"
                "call _sys_write\n\t"
                "addl $8,%%esp\n\t"
                "popl %0\n\t"
                "pop %%fs"
                ::"r" (i),"r" (fd):"ax", "dx");
    }
    else
    {
        if( !( file=task[0]->filp[fd] ) )
            return 0;
        inode=file->f_inode;
        __asm__ ("push %%fs\n\t"
```

```

    "push %%ds\n\t"
    "pop %%fs\n\t"
    "pushl %0\n\t"
    "pushl $_logbuf\n\t"
    "pushl %1\n\t"
    "pushl %2\n\t"
    "call _file_write\n\t"
    "addl $12,%%esp\n\t"
    "popl %0\n\t"
    "pop %%fs"
    :: "r" (i), "r" (file), "r" (inode) );
}
return i; // 返回字符串长度
}

```

可以将该函数定义在 kernel/printk.c 文件中，并且在 include/linux/kernel.h 文件中添加该函数的声明。第一个参数 fd 是文件描述符，为 1 时将信息写入标准输出 stdout，为 3 时将信息写入 process.log 文件；第二个参数 fmt 是格式化字符串，类似于 printf 的第一个参数；后面的是可变参数列表，类似于 printf 的可变参数列表。

3.3.3 记录进程的运行轨迹

之前已经打开了 process.log 文件并解决了向 process.log 文件写入数据的问题，接下来需要解决在什么时刻将进程的运行轨迹信息写入 process.log 文件的问题。这就要求读者对进程的状态转换以及进程调度有一个全面的了解。

Linux 0.11 支持四种经典的进程状态的转换过程：

- **就绪到运行**：通过 schedule 函数（在文件 kernel/sehed.c 的第 135 行）完成。
- **运行到就绪**：通过 schedule 函数完成。
- **运行到睡眠**：通过 sleep_on 函数（在文件 kernel/sehed.c 的第 202 行）和 interruptible_sleep_on 函数（在文件 kernel/sehed.c 的第 225 行）完成。或者通过进程主动睡眠的系统调用内核函数 sys_pause（在文件 kernel/sehed.c 的第 192 行）和 sys_waitpid（在文件 kernel/exit.c 的第 183 行）完成。
- **睡眠到就绪**：通过 wake_up 函数（在文件 kernel/sehed.c 的第 251 行）完成。

此外还有进程的创建和退出两种情况：

- **进程的创建**：通过 copy_process 函数（在文件 kernel/fork.c 的第 89 行）完成。
- **进程的退出**：通过 do_exit 函数（在文件 kernel/exit.c 的第 122 行）完成。

所以，只要在以上提到的这些函数的适当位置调用 fprintk 函数输出日志信息到 process.log 文件，就能完成进程轨迹的全面跟踪了。在调用 fprintk 函数前，先定义 process.log 文件中每行日志的格式为：

```
pid      state      time
```

其中“pid”是进程的 id。“state”表示进程刚刚进入的状态，其取值及意义如下表：

state	意义
N	进程刚刚创建

J	进程进入就绪态
R	进程进入运行态
W	进程进入阻塞态
E	进程退出

“time”表示进程刚刚发生状态转换的时刻。这个时刻不是实际的时间，而是系统的滴答时间 jiffies（参见之后的详细介绍）。这三个字段之间用制表符“\t”分隔。例如“06 J 529”表示进程 6 在第 529 个系统滴答时间进入了就绪状态。

jiffies 滴答

jiffies 是一个全局变量（在文件 kernel/sched.c 的第 72 行定义），它记录了 Linux 0.11 操作系统从开机到当前时刻的 8253 定时计数器发生的中断次数。在 sched_init 函数中（在 kernel/sched.c 文件的第 528 行），定时计数器的中断处理程序被设置为 timer_interrupt 函数，在此函数响应定时计数器中断的过程中，每次都会将 jiffies 的值增加 1（在文件 kernel/system_call.s 的第 243 行）。

此外，在 sched_init 函数中有如下代码（在 kernel/sched.c 文件的第 524 行）：

```
.....
outb_p(0x36, 0x43);
outb_p(LATCH & 0xff, 0x40);
outb(LATCH >> 8, 0x40);
.....
```

这段代码用来初始化 8253 定时计数器，设置中断时间间隔为 LATCH，而 LATCH 是一个宏定义，在文件 kernel/sched.c 的第 57 行定义如下：

```
#define LATCH (1193180/HZ)
```

宏 HZ 在文件 include/linux/sched.h 的第 5 行定义如下：

```
#define HZ 100
```

同时，由于 8253 定时计数器输入时钟频率为 1.193180MHz（即 1193180/每秒），所以，LATCH=1193180/100 就是将定时计数器设置为每跳 11931.80 下产生一次时钟中断，即每 1/100 秒（10 毫秒）产生一次时钟中断。所以，jiffies 实际上记录了从开机以来共经历了多少个 10 毫秒。

下面的表格列出了在 Linux 0.11 的内核源代码中定义的进程状态的名称，在 Linux 0.11 内核项目中，使用“查找和替换”功能，以这些状态名称为关键字进行搜索，就可以找到进程状态发生改变的所有源代码。读者也可以按照下面的步骤逐步添加调用 fprintf 函数的语句：

状态定义	含义
TASK_RUNNING	可运行
TASK_INTERRUPTIBLE	可中断的等待状态
TASK_UNINTERRUPTIBLE	不可中断的等待状态
TASK_ZOMBIE	僵死
TASK_STOPPED	暂停
TASK_SWAPPING	换入/换出

1. 为了跟踪进程的创建，可以修改 kernel/fork.c 文件中的 copy_process 函数。在第 114 行后面增加语句（一定在子进程控制块的 pid 和 start_time 被赋值之后）：

```
p->start_time = jiffies;
```

```
fprintk( 3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies );
```

记录进程刚刚创建并处于不可中断睡眠状态。在第 164 行后面增加语句：

```
fprintk( 3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies );
```

记录刚刚创建的进程进入就绪状态。

2. 当进程的状态被设置为 TASK_ZOMBIE，并且已经获取了退出码时，表示进程已经完成了退出操作，虽然此时进程的控制块还没有被释放（要留待父进程获取子进程的信息）。子进程退出的最后一步是发送信号通知父进程，目的是唤醒正在等待此事件的父进程，由父进程来释放子进程的控制块。从时序上来说，应该是子进程先退出，父进程才被唤醒。所以，为了跟踪进程的退出，可以修改 kernel/exit.c 文件中的 do_exit 函数。在第 159 行将进程状态设置为 TASK_ZOMBIE 的后面插入一条语句，如下：

```
current->state = TASK_ZOMBIE;
```

```
fprintk( 3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies );
```

3. 接下来，修改 kernel/sched.c 文件中的 Schedule 函数，跟踪进程进入就绪状态或运行状态。注意，Schedule 函数找到的 next 进程是接下来将要运行的进程。如果 next 恰好是当前正处于运行态的进程，switch_to(next) 也会被调用，这种情况相当于当前进程的状态没有改变。在第 157 行进程由于收到信号而进入就绪状态，所以需要在此行的后面添加一条语句：

```
fprintk( 3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies );
```

注意此句要跟上面一句代码共同属于 if 语句，需加 {}

在第 188 行 switch_to(next) 语句前添加下面的代码：

```
if(current->state == TASK_RUNNING && current != task[next])
```

```
    fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);
```

```
if(current != task[next])
```

```
    fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);
```

4. 另外，只要操作系统处于空闲状态，就会让进程 0 运行，进程 0 会不停的调用 sys_pause 函数，以激活调度程序，使得调度程序可以随时选择处于就绪态的进程来运行。此时，可以认为进程 0 处于等待状态（等待有其它可运行的进程），也可以认为进程 0 处于运行态，因为它是唯一在 CPU 上运行的进程，只不过运行的效果是等待。所以，在修改 sys_pause 函数时，添加的跟踪语句应该为：（在第 202 行 schedule 函数的前面）

```
if(current->pid != 0)
```

```
    fprintk( 3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies );
```

5. 注意，当进程被唤醒时虽然会将进程设置为 TASK_RUNNING 状态，但是进程实际是进入了就绪状态，所以此时在日志记录中应使用标志 ‘J’。只有在 Schedule 函数的最后，被选中为 next 的进程才会进入实际的运行状态，才能在日志记录中使用标志 ‘R’。也就是说，在读者添加的所有调用 fprintk 函数的语句中，只有在 Schedule 函数的最后这一个地方使用 ‘R’ 标志。余下的跟踪语句请读者在适当的位置自行添加。（提示，包括前面提到的几处，总共需要添加 15 处，可将 “fprintk” 作为 “查找” 功能的关键字进行查找，确保数量正确）。

待读者添加完所有的跟踪语句之后，按照步骤完成下面的操作：

1. 使用在第 3.2 节创建的 Linux011 应用程序项目的硬盘镜像文件 `harddisk.img` 覆盖当前 Linux011 Kernel 项目的 `harddisk.img` 文件，这样就可以继续使用之前生成的应用程序可执行文件 `app` 了。
2. 按 F5 启动调试 Linux011 Kernel 项目。
3. 执行 `app` 可执行文件（在纸上记录下打印输出的父进程和子进程的 `id`，以便下面分析数据）。
4. 将 `process.log` 文件复制到软盘 B，命令如下：

```
mcopy /var/process.log b:log.txt
```
5. 结束此次调试。
6. 在“项目管理器”窗口中双击 `floppyb.img` 文件，使用软盘编辑器工具打开，将软盘 B 中的 `log.txt` 文件复制到 Windows 的 C 盘根目录下。
7. 打开 `log.txt` 文件（可以将 `log.txt` 文件拖动到 Linux Lab 窗口中释放）。要求读者在日志文件中能够找到类似于下面日志的 `app` 应用程序的运行轨迹信息。其中，04 是 Shell 程序的进程 `id`，06 是 `app` 的主进程 `id`，07、08 和 09 是三个子进程 `id`。结合应用程序 `app` 的源代码及其运行时打印输出的信息分析这些日志，理解进程在生命周期中状态的转换过程和进程调度过程。

```

.....
4   J   528
4   R   528
06  N   529
06  J   530
04  W   530
06  R   530
07  N   534
07  J   534
08  N   535
08  J   535
09  N   535
09  J   536
06  W   536
09  R   536
09  E   537
06  J   537
08  R   537
08  E   538
07  R   538
07  E   539
06  R   539
06  E   539
.....

```

在之前的应用程序 `app` 中还无法控制各个子进程运行的时间，特别是无法控制子进程实际占用 CPU 的时间和等待 I/O 操作的时间，也就无法对进程的调度过程进行量化分析。但是，如果在内核中对各种设备的 I/O 操作时间进行统计，并在应用程序中对各种

I/O 设备进行访问也不现实。所以，接下来采用在应用程序中模拟这样一种折中的方式，从而实现对进程调度过程的量化分析。

3.4 编写用于模拟的应用程序

修改在第 3.2 节中编写的多进程程序作为测试程序。首先在 LinuxApp.c 文件中添加一个函数 `cpuio_bound`，用来模拟进程在生命周期中占用 CPU（运行态）与 I/O 操作（阻塞态）的情景，代码为：

```
#include<sys/wait.h>
#include<linux/sched.h>
#include<time.h>
void cpuio_bound( int last, int cpu_time, int io_time )
{
    struct tms start_time, current_time;
    clock_t utime, stime;
    int sleep_time;
    while( last>0 )
    {
        times( &start_time );
        do
        {
            times( &current_time );
            utime=current_time.tms_utime-start_time.tms_utime;
            stime=current_time.tms_stime-start_time.tms_stime;
        }while( ( ( utime+stime )/HZ )< cpu_time );
        last-=cpu_time;
        if( last<=0 )
            break;
        sleep_time=0;
        while( sleep_time<io_time )
        {
            sleep( 1 );
            sleep_time++;
        }
        last-=sleep_time;
    }
}
```

参数 `last` 表示占用 CPU 以及 I/O 操作的总时间，不包括在就绪队列中的时间。参数 `cpu_time` 表示一次连续占用 CPU 的时间，必须大于等于 0。参数 `io_time` 表示一次 I/O 操作占用的时间，必须大于等于 0。如果 `last > cpu_time + io_time`，则往复多次占用 CPU 和 I/O 操作，且时间单位均为秒。其中 `struct tms` 在 `include/sys/times.h` 中定义，`clock_t` 在 `include/time.h` 中定义。将 `main` 函数修改为如下的代码：

```
int main( int argc, char * argv[] )
{
    pid_t p1, p2, p3, p4;
    if( ( p1=fork() )==0 )
```

```

    { printf( "in child1\n" ); cpuio_bound( 5, 2, 2 );}
    else if( ( p2=fork() )==0 )
    { printf( "in child2\n" ); cpuio_bound( 5, 4, 0 );}
    else if( ( p3=fork() )==0 )
    { printf( "in child3\n" ); cpuio_bound( 5, 0, 4 );}
    else if( ( p4=fork() )==0 )
    { printf( "in child4\n" ); cpuio_bound( 4, 2, 2 );}
    else
    {
        printf( "=====This is parent process=====\\n" );
        printf( "pid=%d\\n", getpid() );
        printf( "pid1=%d\\n", p1 );
        printf( "pid2=%d\\n", p2 );
        printf( "pid3=%d\\n", p3 );
        printf( "pid4=%d\\n", p4 );
    }
    wait( NULL );
    return 0;
}

```

按照下面步骤操作，得到最新的日志文件：

1. 按 F7 生成 Linux011 应用程序项目，F5 启动调试。
2. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下
`mcopy b:linuxapp.exe app`
3. 给 app 文件添加可执行权限，命令如下：
`chmod +x app`
4. 执行“sync”命令。
5. 结束调试，用 Linux011 应用程序项目中的硬盘镜像文件 `harddisk.img` 覆盖 Linux011 Kernel 项目中的 `harddisk.img` 文件，这样就可以在内核项目中继续使用刚刚生成的 app 可执行文件了。
5. 按 F5 启动调试 Linux011 Kernel 项目。
6. 执行 app 可执行文件（记录父进程和子进程的 id）。
7. 将 process.log 文件复制到软盘 B，命令如下：
`mcopy /var/process.log b:log.txt`
8. 结束此次调试。
8. 在“项目管理器”窗口中双击 `floppyb.img` 文件，使用软盘编辑器工具打开，将软盘 B 中的 `log.txt` 文件复制到 Windows 的 C 盘根目录下。
9. 打开 `log.txt` 文件（可以将 `log.txt` 文件拖动到 Linux Lab 窗口中释放）。查看日志文件中记录的信息，根据之前记录的进程 id 找出 app 应用程序运行时产生的进程调度轨迹信息。

根据信息以及 app 的源文件思考 app 运行时父进程及子进程在生命周期状态转换的过程，分析进程调度的过程及其原因，进一步理解进程在生命周期中进程的状态转换和进程调度。

3.5 统计并分析数据

前面已经得到 log.txt 日志文件，为了加深对进程调度和调度算法的理解；接下来统计分析 log.txt 文件的数据，计算平均周转时间、平均等待时间和吞吐率，量化分析进程调度和调度算法。分析数据步骤如下：

1. 为了从 log 文件读取数据，然后计算平均周转时间、平均等待时间和吞吐率，需要编写一个程序，用什么语言编写都行，读者可自行设计。为了方便，给读者提供一个 Python 语言编写的程序。打开“学生包”找到本实验对应的文件夹，将其中的“stat_log.py”文件复制到 C 盘根目录下。（在“学生包”本实验对应文件夹下提供了 Python 的安装包，有需要的读者可自行安装到自己的电脑上）
2. 检查 log.txt 日志文件，保证文件的最后一条记录是完整的，若不完整则删除。
3. 启动 Windows 的控制台，将当前目录设置为 C 盘根目录，然后输入命令：

```
C:\>stat_log.py log.txt -g > out.txt
```

如果对 C 盘根目录没有写权限，可以将这些文件移动到其他盘的根目录或文件夹中。

4. 查看在 C 盘根目录下生成的 out.txt 文件中的内容，其中一部分内容如下：

(Unit: tick)

Process	Turnaround	Waiting	CPU Burst	I/O Burst
0	3501	64	8	0
1	1680	0	1	1679
2	24	4	20	0
3	3003	0	4	2999
4	2948	43	45	2860
5	3	1	2	0
6	868	0	7	860
7	1008	398	400	210
8	1208	407	800	0
9	861	0	0	860
10	4	0	3	0
11	520	2	66	452
12	10	0	9	0
Average:	1202.92	70.69		
Throughout:	0.37/s			

意义如下表：

名称	意义
Process	进程 id
Turnaround	周转时间
Waiting	等待时间
CPU Burst	占用 CPU 时间
I/O Burst	I/O 时间
Average	平均周转时间和平均等待时间
Throughout	吞吐率

进程 0 比较特殊，因为其运行轨迹没有被完全记录下来，所以得到的信息误差比较大，可以忽略进程 0；其它进程的信息可能也存在误差，但在误差允许范围之内。

3.6 调度算法、时间片和优先级

根据 Linux 0.11 的进程调度函数 `schedule` 的代码（在文件 `kernel/sched.c` 的第 135 行），可知 Linux 0.11 的调度算法是选取 `counter`（时间片）最大的就绪进程占用 CPU。运行态进程（即 `current`）的 `counter` 每当发生时钟中断时就会减 1（在 `do_timer` 函数中完成，在文件 `kernel/sched.c` 的第 416 行），所以是一种比较典型的时间片轮转调度算法。另外，由 `schedule` 函数可以看出，当没有 `counter` 大于 0 的就绪进程时，要对所有进程做 “`(*p)->counter = ((*p)->counter >> 1) + (*p)->priority;`” 操作（`kernel/sched.c` 第 182 行），效果是对所有的进程（包括阻塞进程）进行 `counter` 的衰减（除 2），并累加 `priority` 的值，这样对已经阻塞的进程来说，一个进程在阻塞队列中停留的时间越长，其优先级越大，被分配的时间片就越大（不会大于优先级的 2 倍），而对于时间片已经为 0 的进程来说，其时间片的值会被重置为其优先级的值。所以总的来说，Linux 0.11 的进程调度是一种综合考虑进程优先级，并能动态反馈调整时间片的轮转调度算法。

再来看一下进程的时间片是如何被初始化的。进程在被创建时，在 `copy_process` 函数（`kernel/fork.c` 第 89 行）中有如下代码：

```
.....
*p = *current;
.....
p->counter = p->priority;
.....
```

虽然父进程的时间片会发生变化，但是优先级不会改变，上面的第二句代码将 `p->counter` 设置成 `p->priority`，说明在创建进程时，分配的时间片是一个固定值。同时，由于每个进程的优先级都是从父进程继承的，除非自己通过调用 `nice` 系统调用函数（在文件 `kernel/sched.c` 的第 506 行）修改优先级。结合以上的因素，如果没有调用 `nice` 系统调用，进程时间片的初值就是进程 0 的优先级。进程 0 的优先级由宏 `INIT_TASK`（在文件 `include/linux/sched.h` 的第 176 行）定义如下：

```
#define INIT_TASK \
{ 0, 15, 15, \ //state, counter, priority
.....
```

所以，如果修改了进程 0 的优先级，就会修改所有进程的初始时间片。

现在请读者尝试将进程的初始时间片改大一些（大于 15），重新按照 3.4 的内容跟踪进程运行的轨迹，得到新的日志文件，然后按照 3.5 的内容得到统计数据。再请读者尝试将进程的初始时间片改小一些（小于 15），同样得到统计数据。最后，请读者将得到的统计数据填入下面的表格，通过对比这些数据体会时间片大小对平均周转时间、平均等待时间和吞吐率的影响，并分析其中的原因。

时间片大小	平均周转时间	平均等待时间	吞吐率
一个小于 15 的值			
15			
一个大于 15 的值			
更多值			

实验七 信号量的实现和应用

实验性质：验证、设计

建议学时：2 学时

实验难度：★★★★☆

一、实验目的

- 加深对进程同步与互斥概念的理解。
- 掌握信号量的使用方法，并解决生产者—消费者问题。
- 掌握信号量的实现原理。

二、预备知识

信号量

信号量（semaphore）最早由荷兰科学家、图灵奖获得者 E.W. Dijkstra 设计。Linux 的信号量遵循 POSIX 规范，可以使用“man sem_overview”命令查看相关信息。但 Linux 0.11 还没有实现信号量，也不支持“man”命令。Linux 0.11 是一个支持多进程并发的现代操作系统，虽然它还没有为应用程序提供任何锁或者信号量，但在其内核部分已经通过关中断、开中断的方式实现了锁机制，这样就允许执行原子操作，即在多个进程访问共享的内核数据时用来实现互斥和同步。通过使用 Linux 0.11 内核提供的锁机制，就可以实现信号量。本次实验涉及到的信号量系统调用可以参见下面的表格：

函数原型	说明
sem_t* sem_open(const char* name, unsigned int value)	创建/打开信号量。name 就是信号量的名字，不同的进程可以通过提供同样的 name 而共享一个信号量。value 是信号量的初始值，仅当新建信号量时，此参数才有效。
int sem_wait(sem_t* sem)	等待信号量。就是信号量的 P 原子操作。如果继续运行的条件不满足，则令调用进程等待在信号量 sem 上。
int sem_post(sem_t* sem)	释放信号量。就是信号量的 V 原子操作。如果有等待 sem 的进程，它会唤醒其中的一个。
int sem_unlink(const char* name)	关闭名字为 name 的信号量

生产者—消费者问题

生产者—消费者问题是一个著名的进程同步问题。它描述的是：有一群生产者进程在生产某种产品，并将此产品提供给一群消费者进程去消费。为使生产者进程和消费者进程能并发执行，在他们之间设置了一个具有 n 个缓冲区的缓冲池，生产者进程可以将它生产的一个产品放入一个缓冲区中，消费者进程可以从一个缓冲区中取得一个产品消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品，也不允许生产者进程向一个已经装有产品的缓冲区中放入产品。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。

3.2 在内核中实现信号量的系统调用

按照下面的内容在 Linux 0.11 内核项目中实现简易版的信号量。只需要修改 include/unistd.h、kernel/system_call.s 和 include/linux/sys.h 三个文件、实现信号量的四个系统调用函数 sem_open、sem_wait、sem_post 和 sem_unlink 即可：

1. 在文件 include/unistd.h 中的第 161 行之后，定义四个新的系统调用号，如下：

```
#define __NR_sem_open 87
#define __NR_sem_wait 88
#define __NR_sem_post 89
#define __NR_sem_unlink 90
```

2. 在文件 kernel/system_call.s 中的第 73 行，修改系统调用的总数：

```
nr_system_calls = 91
```

3. 在文件 include/linux/sys.h 中的第 87 行之后，添加系统调用内核函数的声明：

```
extern int sys_sem_open();
extern int sys_sem_wait();
extern int sys_sem_post();
extern int sys_sem_unlink();
```

在此文件的最后，向系统调用函数指针表 sys_call_table[] 中添加新系统调用函数的指针（注意，系统调用号必须与系统调用内核函数指针在系统调用函数表中的索引一一对应），如下：

```
fn_ptr sys_call_table[] = {
.....
sys_sem_open,      //87
sys_sem_wait,      //88
sys_sem_post,      //89
sys_sem_unlink     //90
};
```

4. 打开“学生包”文件夹，在本实验对应的文件夹中找到“sem.c”文件。将此文件拖动到 Linux Lab 中释放，即可打开此文件。其中实现了信号量的四个系统调用函数。
5. 在 Linux Lab 的“项目管理器”窗口中，右键点击“kernel”文件夹节点，选择菜单“添加”中的“添加新文件”，打开“添加新文件”对话框。
6. 在“添加新文件”对话框中选中“C 源文件(.c)”模板，在输入文件名称“semaphore.c”后点击“添加”按钮，添加一个新的 C 源文件 semaphore.c。
7. 将 sem.c 中的源代码复制到刚刚创建的 semaphore.c 文件中。
8. 按 F7 生成项目，确保没有语法错误。

现在已经在 Linux 0.11 的内核中添加了信号量相关的系统调用函数，在应用程序中就可以使用信号量完成进程的同步和互斥操作了。

仔细阅读文件 semaphore.c 中的源代码及注释，重点理解下面的内容：

● sem_t 结构体

用来定义信号量。成员 sem_name 是信号量的名称字符串。成员 value 是信号量中可用资源的数量。成员 used 是信号量的引用计数，每当有进程打开信号量时加 1，每当有进程关闭信号量时减 1，为 0 时表示信号量未被使用。全局变量 sem_array

用来存放信号量的结构体数组，本实验中将要用到的“empty”、“full”、“mutex”信号量都会保存在此数组中。

- **item 链表项**

此链表项可用来链接成一个存放进程（任务）的链表。sem_t 结构体中的成员 wait 是此链表的头，作为存储阻塞在信号量上进程的链表，并遵循先来先服务的原则。

- **get_name 函数和 find_sem 函数**

get_name 函数的作用是将用户传入的字符串参数复制到内核，再返回内核中字符串的地址。find_sem 函数的作用是在全局变量 sem_array 中查找是否存在与传入的字符串参数同名的信号量。

- **信号量的PV原语操作**

原语操作的实现方式可以参考本书第2章的第4节。

P原语操作由函数sys_sem_wait实现，动作如下：

- 1) 信号量的value减1；
- 2) 若value减1后仍大于等于零，则进程继续执行；
- 3) 若value减1后小于零，则将进程放入该信号量的等待队列，并使之进入阻塞状态，最后执行进程调度。

V原语操作由函数sys_sem_post实现，动作如下：

- 1) 信号量的value加1；
- 2) 若value大于零，则进程继续执行；
- 3) 若value小于等于零，则从该信号量的等待队列中唤醒一个进程，然后再返回该进程继续执行或执行进程调度。

- **cli 与 sti函数**

实现原语操作时调用了cli和sti函数。其中，cli 禁止中断发生，sti 允许中断发生。

3.3 在 Linux 应用程序中使用信号量解决生产者—消费者问题

打开“学生包”，在本实验文件夹中提供了使用信号量解决生产者—消费者问题的参考源代码文件 pc.c。将其拖放到 Linux Lab 即可打开，仔细阅读此文件中的源代码和注释，注意以下几点：

- 本实验是用文件作为生产者和消费者之间的共享缓冲区的。当生产者在文件中写入产品数据后，必须调用fflush函数，将磁盘缓冲区中的内容真正写入磁盘，才能确保消费者读取到正确的产品数据。
- fork函数调用成功后，子进程会继承父进程拥有的大多数资源，包括父进程打开的文件，所以子进程可以直接使用父进程已经打开的文件。
- 当多个进程同时使用printf函数向终端输出信息时，终端也成为了一个临界资源，需要做好互斥保护，否则输出的信息可能错乱。另外，调用printf函数之后，信息只是保存在内核的输出缓冲区中，还没有真正送到终端上显示，这也可能造成输出信息顺序不一致，所以，必须使用函数fflush(stdout)确保数据送到终端。
- 生产者和消费者每次循环的最后都调用了 sleep 函数，从而确保生产产品和消费产品的速度差异，才会产生同步、互斥的效果。

按照下面的步骤查看生产者——消费者同步执行的过程：

1. 双击 Linux Kernel 项目左侧“项目管理器”窗口中的 floppyb.img 文件节点，会自动使用 Floppy Image Editor 工具打开此软盘镜像文件，将 pc.c 文件拖动到此工具窗口中释放，点击工具栏上的保存按钮后关闭此工具。

2. 按 F5 键启动调试，待 Linux 0.11 启动后，将软盘 B 中的 pc.c 文件拷贝到硬盘的当前目录，命令如下：

```
mcop y b:pc.c pc.c
```
 3. 使用 gcc 编译 pc.c 文件，命令如下：

```
gcc pc.c -o pc
```
 4. 执行 sync 命令将对硬盘的更改保存下来。
 5. 执行 pc 命令，查看生产者—消费者同步执行的过程，如图 7-1 所示。
 6. 由于 Linux 0.11 不支持向上滚屏查看，可使用命令“pc > pc.txt”将输出保存到文件 pc.txt 中，再使用命令“mcop y pc.txt b:pc.txt”将其复制到软盘 B。
 7. 停止此次调试。在“项目管理器”窗口中双击 floppyb.img 文件节点，将软盘 B 中的 pc.txt 文件复制到 Windows 的任意文件夹中，再拖放到 Linux Lab 打开即可。
- 仔细观察文件 pc.txt 中的执行结果，并思考下面的问题：
- 生产者和消费者是如何使用 Mutex、Empty 和 Full 信号量实现同步的？在两个进程中对这三个同步对象的操作能够改变顺序吗？
 - 生产者在生产了 13 号产品后本来要继续生产 14 号产品，可此时生产者为什么必须等待消费者消耗一个产品后才能生产 14 号产品呢？生产者和消费者是怎么样使用信号量实现该同步过程的呢？

```

Producer 15 : 00 at 0
Producer 15 : 01 at 1
Producer 15 : 02 at 2
Producer 15 : 03 at 3
Producer 15 : 04 at 4
Producer 15 : 05 at 5
Producer 15 : 06 at 6
Producer 15 : 07 at 7
Producer 15 : 08 at 8
Producer 15 : 09 at 9
Producer 15 : 10 at 0
Producer 15 : 11 at 1
Producer 15 : 12 at 2
Producer 15 : 13 at 3
Producer 15 : 14 at 4
Producer 15 : 15 at 5

Consumer 16: 00 at 0
Consumer 16: 01 at 1
Consumer 16: 02 at 2
Consumer 16: 03 at 3
Consumer 16: 04 at 4
Consumer 16: 05 at 5
Consumer 16: 06 at 6

```

图 7-1 生产者—消费者同步执行的过程

3.4 调试信号量的工作过程

3.4.1 创建信号量

1. 在文件 kernel/semaphore.c 的 sys_sem_open 函数中调用 cli 函数处(第 90 行)添加一个断点。
2. 按 F5 启动调试，执行 pc 命令，会命中此断点。按 F10 执行 cli 函数后，再按 F10 执行调用了 get_name 函数的代码行，get_name 函数的作用是将用户空间输入的字符串参数 name 复制到内核空间中，此时查看变量 kernel_name 的值应为

“empty”，说明应用程序 pc 正在创建名称为“empty”的信号量，这与应用程序 pc 中创建信号量的顺序是一致的。

3. 再按 F10 单步调试一行代码，会执行调用了 find_sem 函数的代码行。find_sem 函数在信号量数组 sem_array 中查找是否有与当前要创建的信号量同名的信号量，显然此时还没有“empty”信号量。所以，黄色箭头会在 for 循环处（第 103 行）停止。此循环体的功能是在信号量数组 sem_array 中找到一个还未被使用的信号量，并初始化其成员，使之作为新创建的信号量。
4. 继续按 F10 单步调试，直到从第 112 行返回。调试的过程中注意观察信号量的各个成员是如何被初始化的。
5. 按 F5 继续运行，仍然会在之前添加的断点处中断，可以按照之前的步骤继续调试“full”和“mutex”信号量的创建过程。

3.4.2 等待、释放信号量

消费者等待“full”信号量（阻塞）

生产者和消费者刚开始执行时，用来存放产品的缓冲区是空的，所以消费者在第一次调用 sem_wait 函数等待 full 信号量时，应该会阻塞。按照下面的步骤调试：

1. 结束之前的调试，删除所有断点。
2. 在文件 kernel/semaphore.c 的 sys_sem_wait 函数中调用 cli 函数处（第 123 行）添加一个断点。
3. 按 F5 启动调试，执行 pc 命令，会命中此断点。此函数中用到的 current 全局变量总是指向当前正在运行的进程。使用“快速监视”功能查看表达式“*current”的内容，查看其成员 pid 的值，应该与 Bochs Display 窗口中此时输出的消费者进程的 ID 值相同。使用“快速监视”功能查看表达式“*sem”的内容，可以确定其为“full”信号量。
4. 多次按 F10 单步调试，直到黄色箭头指向第 129 行。观察调试过程可以看到，消费者进程在将“full”信号量的资源减 1 后，调用 wait_task 函数将自己添加到了“full”信号量的等待任务队列中，并进入阻塞状态，最后执行调度程序让出处理器，从而使生产者进程可以获得处理器。

生产者等待“empty”信号量（不阻塞）

1. 按 F5 继续执行，仍然会在之前添加的断点处中断。使用“快速监视”功能查看表达式“*current”的内容，查看其成员 pid 的值，应该与 Bochs Display 窗口中此时输出的生产者进程的 ID 值相同。使用“快速监视”功能查看表达式“*sem”的内容，可以确定其为“empty”信号量。
2. 多次按 F10 单步调试，直到黄色箭头指向第 132 行。可以看到，由于“empty”信号量中有足够的资源（所有缓冲区都为空），所以生产者线程并没有阻塞，而是在消耗了一个资源后直接返回了。

生产者释放“full”信号量（唤醒消费者进程）

1. 删除之前添加的断点，在文件 kernel/semaphore.c 的 sys_sem_post 函数中调用 cli 函数处（第 141 行）添加一个断点。
2. 按 F5 继续执行，在生产者释放“mutex”信号量时会命中此断点，再按 F5 继续执行，会再次命中此断点。此时，使用“快速监视”功能查看表达式“*current”的内容，查看其成员 pid 的值，应该与生产者进程的 ID 值相同。使用“快速监视”功能查看表达式“*sem”的内容，可以确定其为“full”信号量。Bochs Display 窗口中显示生产者已经生产了一个产品。
3. 多次按 F10 单步调试，直到黄色箭头指向第 154 行。使用“快速监视”功能查看表

达式“*p”的内容，查看其成员 pid 的值，应该与消费者进程的 ID 值相同，说明生产者在将“full”信号量的资源数量加 1 后，将阻塞在“full”信号量等待队列队首的消费者进程唤醒（使之进入就绪状态并移出等待队列）。

4. 消费者进程被唤醒后，会从前被阻塞的位置（第 129 行）继续执行，所以在第 131 行添加一个断点，按 F5 继续调试会命中此断点。此时，使用“快速监视”功能查看表达式“*current”的内容，查看其成员 pid 的值，应该与消费者进程的 ID 值相同。使用“快速监视”功能查看表达式“*sem”的内容，可以确定其为“full”信号量，其资源数量已经恢复为 0，等待队列也为空。

消费者释放“empty”信号量（不唤醒）

1. 删除第 131 行的断点，保留第 141 行的断点，按 F5 继续执行。在消费者释放“mutex”信号量时会命中断点，再按 F5 继续执行，再次命中此断点。此时，使用“快速监视”功能查看表达式“*current”的内容，查看其成员 pid 的值，应该与消费者进程的 ID 值相同。使用“快速监视”功能查看表达式“*sem”的内容，可以确定其为“empty”信号量。Bochs Display 窗口中显示消费者已经消费了一个产品。
2. 多次按 F10 单步调试，直到黄色箭头指向第 154 行。观察调试过程可以看到，消费者在将“empty”信号量的资源数量加 1 后，直接返回了。

请读者删除所有的断点，然后在第 127 行添加一个断点后按 F5 继续调试，观察生产者和消费者执行的过程，直到在刚刚添加的断点处中断。请说明此时是哪个进程在哪个信号量上被阻塞，并分析原因。最后，请读者在关闭信号量的内核函数 sys_sem_unlink 的开始位置添加一个断点，调试一下关闭信号量的过程。

3.5 实现一个生产者进程与多个消费者进程同步工作

修改 pc.c 文件中的源代码，将其改写成生产者进程与多个消费者进程同步工作的程序。将程序运行的结果使用“>>”重定向到文本文件中。

提示：

- 可使用 for 语句循环创建多个消费者进程。
- 本实验使用文件 filebuffer.txt 作为生产者和消费者之间的共享缓冲区，在文件中用于放置产品的共享缓冲区的后面再增加一个计数值（4 个字节），用来记录每次消费者要从该文件中取产品的位置。每个消费者取产品前，先从文件中读取此计数值，消费一个产品后就将此计数值加 1，并重新写入文件的末尾。
- 在父进程结束前要使用 waitpid 函数等待所有消费者进程和生产者进程都结束。

四、思考与练习

1. 本实验的设计者在第一次编写生产者—消费者程序的时候，是这么做的：

<pre> Producer () { P(Mutex); //互斥信号量 生产一个产品item; P(Empty); //空闲缓存资源 将item放到空闲缓存中; V(Full); //产品资源 V(Mutex); } </pre>	<pre> Consumer () { P(Mutex); P(Full); 从缓存区取出一个赋值给item; V(Empty); 消费产品item; V(Mutex); } </pre>
---	--

这样可行吗？如果可行，那么它和标准解法在执行效果上会有什么不同？如果不可行，那么它有什么问题使它不可行？

2. 本实验设计的生产者--消费者问题是在同一个应用程序（同一个main函数）中实现的，请读者试着将生产者和消费者在两个不同的应用程序中实现。

提示：

分别在两个Linux应用程序项目中分别实现生产者程序和消费者程序，生成各自的项目从而得到应用程序的可执行文件，此时，应用程序的可执行文件已经自动写入各自项目文件夹中的软盘镜像文件floppyb.img中了。

将消费者项目文件夹下的floppyb.img拷贝覆盖已经实现了信号量功能的Linux Kernel项目文件夹下的floppyb.img文件，在内核项目中按F5启动调试后，按顺序执行命令：

```
mcopy b:linuxapp.exe consumer
chmod +x consumer
sync
```

从而将消费者程序复制到硬盘，然后结束调试。

将生产者项目文件夹下的floppyb.img拷贝覆盖本已经实现了信号量功能的Linux Kernel项目文件夹下的floppyb.img文件，在内核项目中按F5启动调试后，按顺序执行命令：

```
mcopy b:linuxapp.exe producer
chmod +x producer
sync
```

从而将生产者程序复制到硬盘。

这样在内核项目的硬盘上就同时存在了生产者和消费者应用程序的可执行文件。先执行命令：

```
consumer &
```

此命令会让一个消费者进程在后台开始执行，再执行3次此命令，然后再执行命令：

```
producer
```

这样就可以查看1个生产者和4个消费者同步运行的结果了。（可以将输出结果保存到文件中查看，具体操作步骤可参考本实验3.3的步骤6，但此处输出到文件需要使用“>>”符号在文件的尾部追加写入，如果使用’>’符号，每次都会从文件开始处写入，则无法看到正确的执行结果。另外，“>>”需要写到“&”的前面，这样才能先重定向到文件再在后台运行）

3. 本实验中提供的实现信号量的源代码中没有使用Linux 0.11内核提供的sleep_on函数和wake_up函数。sleep_on函数的功能是将当前进程睡眠在参数指定的链表上，而这个链表是个存在于堆栈中的隐式链表，不太容易被理解。同时，wake_up函数的功能是唤醒隐式链表上睡眠的所有进程，而本实验要求的是仅唤醒等待队列中的一个进程。请读者参考kernel/blk_drv/ll_rw_blk.c文件中第59行的lock_buffer和第69行的unlock_buffer这两个函数，尝试将本实验改成使用sleep_on和wake_up函数来实现信号量的阻塞和唤醒。修改完成后，查看1个生产者和4个消费者同步运行的结果与之前的结果有何不同。根据实验六的内容查看生产者和消费者进程运行的轨迹，可以帮助读者分析其原因（提示：sleep_on函数中用到的隐式链表是将被阻塞的进程插入到其头部，遵循的是后来先服务的原则）。

实验八 地址映射与内存共享

实验性质：验证、设计

建议学时：2 学时

实验难度：★★★★☆

一、实验目的

- 深入理解操作系统的段、页式内存管理。包括理解段表、页表，以及逻辑地址、线性地址、物理地址的映射过程。
- 查看二级页表映射信息，理解页目录和页表的管理方式。
- 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

二、预备知识

请读者阅读《Linux内核完全注释》第4章的前4节，第5.3节以及第13章的内容。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。

3.2 查看物理存储器的信息

请读者按照下面的步骤在Linux011 Kernel项目中添加一个系统调用函数，该函数用来在终端设备上显示物理存储器的信息，包括物理页总数、空闲页数量和占用页数量。

1. 打开“学生包”，在本实验对应文件夹下找到mem.c文件，拖动到Linux Lab中释放，即可打开此文件。将其中的函数physical_mem复制到Linux011 Kernel项目下的mm/memory.c 文件中的末尾处，并且需在include/linux/kernel.h中添加函数的声明。physical_mem函数中的代码比较简单，请读者结合其中的注释自行理解。
2. 添加一个系统调用号为87的系统调用（添加系统调用的方法请参考实验四），该系统调用的内核函数int dump_physical_mem可以写在 kernel/sys.c 文件的末尾，在此函数中直接调用mm/memory.c文件中的physical_mem函数即可。
3. 按F7生成项目，确保没有语法错误和警告。
4. 按F5启动调试，待Linux 0.11完全启动后，使用vi编辑器新建一个main.c文件，其源代码如下所示：

```
#define __LIBRARY__
#include <unistd.h>
#define __NR_dump_physical_mem 87
_syscall0(int, dump_physical_mem)

int main()
{
    dump_physical_mem();
    return 0;
}
```

5. 保存main.c文件并退出vi编辑器后，依次执行如下命令：

```
gcc main.c -o mem
sync
mem
```

应用程序执行后打印输出的物理存储器的信息如图8-1所示：

```
Page Count : 3840
Memory Count : 3840 * 4096 = 15728640 Byte

Free Page Count : 2968
Used Page Count : 872
```

图8-1 物理存储器的使用情况

接下来对 `physical_mem` 函数中的源代码进行修改，查看分配一个物理页和回收一个物理页后物理存储器的变化。

1. 结束之前的调试。
2. 打开“学生包”，在本实验对应文件夹下找到 `mem2.c` 文件，拖动到 Linux Lab 中释放，即可打开此文件。使用其中的函数 `physical_mem` 替换之前添加的 `physical_mem` 函数。
3. 按 F5 启动调试，待 Linux 0.11 完全启动后，执行 `mem` 应用程序，运行结果如图 8-2 所示。请读者理解 Linux 内核函数 `get_free_page` 和 `free_page` 的功能和用法。

```
Page Count : 3840
Memory Count : 3840 * 4096 = 15728640 Byte

Free Page Count : 2968
Used Page Count : 872
*****After Allocate One Page*****
Free Page Count : 2967
Used Page Count : 873
*****After Free One Page*****
Free Page Count : 2968
Used Page Count : 872
```

图 8-2 分配一个物理页和释放一个物理页后物理存储器的变化情况

3.3 跟踪逻辑地址、线性地址、物理地址的映射过程

在学习了 Linux 0.11 管理物理存储器的方式后，接下来重点理解一下从逻辑地址到线性地址，再到物理地址的映射过程。简要过程如下：

首先使用 Bochs 虚拟机的 Debug 功能启动 Linux 0.11，然后运行一个应用程序，其 `main` 函数中主要包括了一个无限循环，源代码如下：

```
#include <stdio.h>
int i = 0x12345678;
int main(void)
{
    printf("The logical/virtual address of i is 0x%08x\n", &i);
    fflush(stdout);
    while(i)
        ;
    return 0;
}
```

应用程序开始运行后，可以使用 Bochs 调试器让应用程序暂停，然后通过使用 Bochs 提供

的调试命令，从变量*i*的逻辑地址计算出它的线性地址，再计算出物理地址。最后，直接修改物理内存让变量*i*的值变为0，从而结束死循环使应用程序退出。请读者按照下面的详细步骤进行实验。

3.3.1 使用 Bochs Debug 作为远程目标机

按照下面的步骤将调试时使用的远程目标机修改为 Bochs Debug:

1. 在 Linux Lab 的“项目管理器”窗口中右键点击项目节点，在快捷菜单中选择“属性”。
2. 在弹出的“属性页”对话框右侧的属性列表中找到“远程目标机”属性，将此属性值修改为“Bochs Debug”。
3. 单击“确定”按钮关闭“属性页”对话框。注意，此时就不能再使用 Linux Lab 提供的可视化调试功能来调试 Linux 0.11 内核了，而只能使用 Bochs 提供的调试命令进行调试。

3.3.2 启动应用程序并暂停

按照下面的步骤启动应用程序，然后使之暂停:

1. 按 F5 启动调试。此时 Bochs 的显示窗口中无内容，而 Bochs 的命令窗口显示将要执行的 BIOS 的第一条指令，并等待用户输入调试命令，如下图:

```
[0xffffffff] f000:fff0 (unk. ctxt): jmp far f000:e05b ; ea5be000f0
<bochs:1>
```

图 8-3 Bochs 的命令窗口等待用户输入调试命令

2. 输入命令“c”按回车，Linux 0.11 会继续运行直到等待用户输入命令。
3. 在 Linux 0.11 中使用 vi 编辑器新建 loop.c 文件，并输入之前给出的源代码。
4. 使用 GCC 将 loop.c 编译为 loop 可执行文件并运行，会打印如下信息:

```
The logical/virtual address of i is 0x00003004
```

图 8-4 loop.c 的执行结果

只要 loop 应用程序不发生改变，0x00003004 这个值在任何一个机器上都是一样的。即使在同一个机器上多次运行，也是一样的。

5. 在 Bochs 的命令窗口(标题为“Bochs for Windows - Console”)中按“Ctrl+c”，Bochs 会暂停运行，进入调试状态。绝大多数情况下都会停在应用程序的死循环内，显示类似如下信息:

```
<0> [0x00faa063] 000f:0000000000000063 (unk. ctxt): cmp dword ptr ds:0x3004, 0x00000000 ; 833d043000000000
```

图 8-5 Bochs 命令窗口在暂停运行后显示的信息

其中的“000f”如果是“0008”，则说明中断在了内核里。那么就要在 Bochs 的命令窗口中输入命令“c”后回车，然后再按“Ctrl+c”，直到变为“000f”为止。如果显示的将要执行的下一条指令不是“cmp ...”，就用“n”命令单步运行几步，直到停在“cmp ...”指令处。

6. 使用命令“u /8”，显示从当前位置开始的 8 条指令的反汇编代码，如下:

```
<bochs:5> u/8
10000063: <                                >: cmp dword ptr ds:0x3004, 0x00000000 ; 833d0430
00000000
1000006a: <                                >: jz  .+0x00000004 ; 7404
1000006c: <                                >: jmp .+0xffffffff5 ; ebf5
1000006e: <                                >: add byte ptr ds:[eax], al ; 0000
10000070: <                                >: xor eax, eax ; 31c0
10000072: <                                >: jmp .+0x00000000 ; eb00
10000074: <                                >: leave ; c9
10000075: <                                >: ret ; c3
```

图8-6 从当前位置开始的8条指令的反汇编代码

这就是loop.c中从while开始一直到return的汇编代码。变量i保存在ds:0x3004这个地址，并不停地和0进行比较，直到它为0，才会跳出循环。

3.3.3 通过段表（GDT和LDT）将逻辑地址映射为线性地址

现在，已经找到了变量i的逻辑地址ds:0x3004。其中，DS寄存器中存储了一个段选择符（也叫段选择子），可以通过这个段选择符在对应的段描述符表中定位一个段描述符，然后再从段描述符中获得段基址，最后让段基址与偏移0x3004相加，就可以获得变量i的线性地址了，如下图所示：

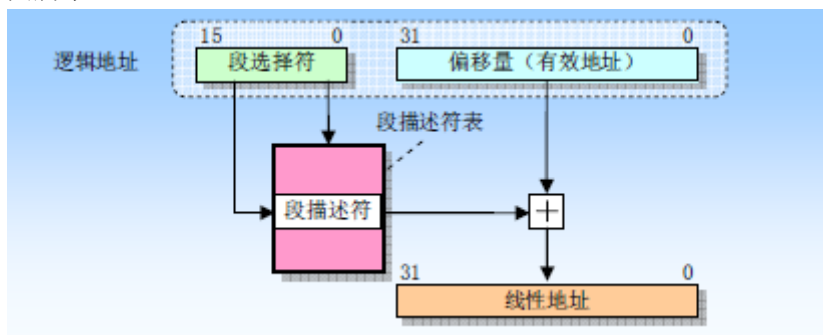


图8-7 逻辑地址到线性地址的变换过程

在开始将逻辑地址映射为线性地址前，需要先掌握段选择符和段描述符的格式。

段选择符

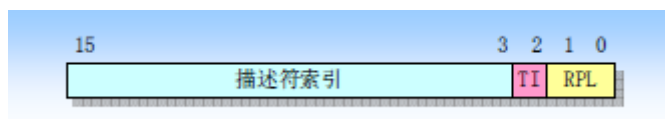


图8-8 段选择符结构

段选择符的大小是2个字节（16位），会被装入CS、DS、SS等段寄存器中，所以和这些段寄存器的大小是一样的。其中，RPL占用最低的2位，表示请求特权级，不参与地址映射的过程。第2位是TI标志，如果TI为0，表示段描述符在GDT（全局描述符表）中，如果TI为1，表示段描述符在LDT（局部描述符表）中。第3到15位表示段描述符在GDT或LDT中的索引。

段描述符

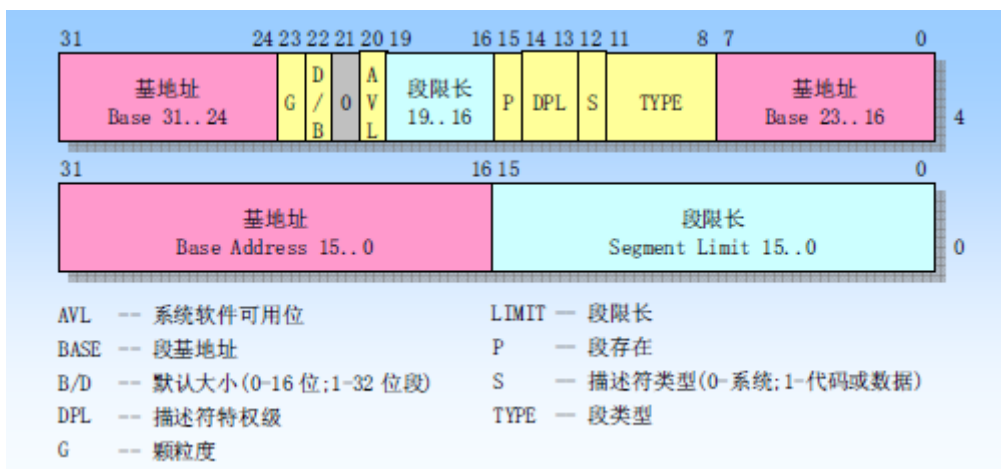


图8-9 段描述符通用格式

段描述符是一个64位的二进制数据，主要存放了段基址和段限长等信息。其中 段基址占

用了32位，由于这32位是分离的，所以需要按照顺序组装起来才能得到完整的段基址；段限长占用了16位；位P (Present) 用来表示段是否存在；位S用来表示段是系统段描述符 (S=0) 还是代码或数据段描述符 (S=1)；4位 TYPE用来表示段的类型，如数据段、代码段、可读、可写等；两位DPL是段的权限，和CPL、RPL对应使用；位G是段限长的粒度，G为0表示段限长以位为单位，G为1表示段限长以4KB为单位；其他内容就不详细解释了。

在了解了段选择符和段描述符后，就可以按照下面的步骤操作了：

1. 在Bochs的命令窗口中输入命令“sreg”查看DS寄存器的值，如下图：

```
<bochs:6> sreg
cs:s=0x000f, dl=0x00000002, dh=0x10c0fa00, valid=1
ds:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=3
ss:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
es:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
fs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
gs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
ldtr:s=0x0068, dl=0x82d00068, dh=0x000082fc, valid=1
tr:s=0x0060, dl=0x82e80068, dh=0x00008bfc, valid=1
gdtr:base=0x00005cb8, limit=0x7ff
idtr:base=0x000054b8, limit=0x7ff
```

图8-10 段寄存器内的值

DS的值为0x0017，换算为16位的二进制为0000000000010111。其中RPL的值为二进制11，十进制为3，是最低的特权级（因为在应用程序中执行）；TI的值为1，表示查找LDT表；使用粗体表示的就是索引值，二进制为10，换算为十进制为2，表示找LDT表中的第3个段描述符（从0开始编号）。

2. 由于LDT表的基址也是由一个段描述符来描述的，而且这个段描述符存储在GDT表中，其索引由ldtr寄存器确定。所以，从上图中也可以得到ldtr的值为0x0068，换算为16位的二进制为0000000001101000，使用粗体表示的就是索引值，二进制为1101，换算为十进制为13，表示LDT表的描述符在GDT的索引为13（第14个描述符）。
3. GDT的起始物理地址存储在寄存器gdtr中，在上图中显示寄存器gdtr的值为0x00005cb8。所以在Bochs的命令窗口中输入命令“xp /2w 0x00005cb8+13*8”，就可以查看GDT表中索引值为13的段描述符了，如下图：

```
0x00000000000005d20 <bogus+ 0>: 0x82d00068 0x000082fc
```

图8-11 GDT表中索引值为13的段描述符

上面两个命令得到的数值可能和这里给出的示例不一致，这是很正常的。如果读者需要确认自己得到的值是否正确，可以查看“sreg”命令输出内容中的ldtr所在行的dl和dh的值，这两个值是x86处理器为了加快地址映射的速度而存储的段描述符，读者得到的段描述符的必须和它们一致。

4. 将段描述符“0x82d00068 0x000082fc”中的加粗部分组合为“0x00fc82d0”，就是LDT表的物理地址了。在Bochs的命令窗口中输入命令“xp /8w 0x00fc82d0”，可以得到LDT表中前4个段描述符的内容，如下图：

```
0x00000000000fc82d0 <bogus+ 0>: 0x00000000 0x00000000 0x000000
02 0x10c0fa00
0x00000000000fc82e0 <bogus+ 16>: 0x00003fff 0x10c0f300 0x000000
00 0x00fc9000
```

图8-12 LDT表中前4个段描述符

第3个段描述符为“0x00003fff 0x10c0f300”就是ds对应的段描述符了。在

“sreg”命令输出的内容中，ds所在行的d1和dh值应该与这个段描述符一致。

段描述符“0x00003fff 0x10c0f300”中加粗部分组合成的“0x10000000”这就是ds段在线性地址空间中的起始地址。用同样的方法也可以计算其它段的基址，都是这个数。段基址+段内偏移就是线性地址了。所以ds:0x3004的线性地址就是0x10000000+0x3004 = 0x10003004。在Bochs的命令窗口中输入命令“calc ds:0x3004”可以查看这个逻辑地址的线性地址，从而可以验证这个结果。

3.3.4 通过页目录和页表将线性地址映射为物理地址

现在，已经得到了变量i的线性地址0x10003004，使用线性地址最高10位的值作为页目录中的索引（页目录号），就可以得到页表的起始物理地址，再使用线性地址中间10位的值作为页表中的索引（页表号），就可以找到物理页的起始地址，最后使用线性地址最低12位的值作为物理页内的偏移（页内偏移），就可以确定最终的物理地址了。如下图所示：

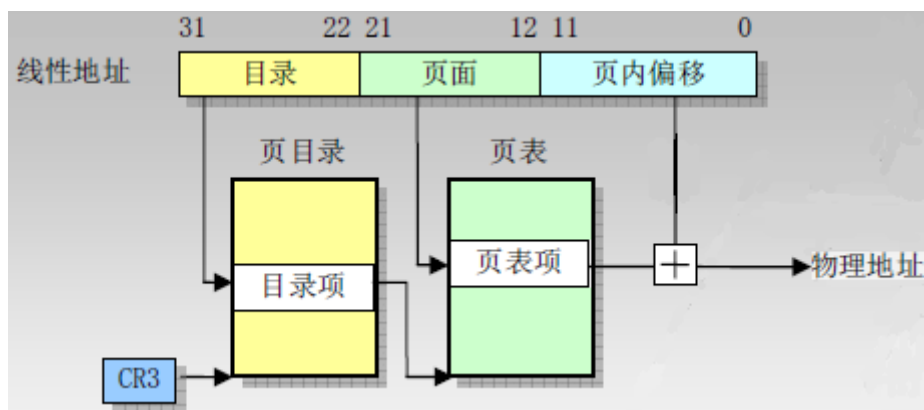


图8-13 线性地址到物理地址的变换过程

在开始将线性地址映射为物理地址前，需要先掌握页目录和页表的格式。

页目录和页表

页目录和页表的格式是一样的，大小均为4KB，包含1024个页表项，每项有4个字节（32位）。如下图所示：



图8-14 页目录和页表中的表项的格式

每个页表项最高的20位是物理页框号，将物理页框号左移12位可以得到其对应物理页的基址，所以每个物理页的大小也是4KB。页表项中余下的12位是属性位，读者可以自己了解。一个进程的线性地址空间通常由一个页目录和多个页表来描述，页目录中的页表项用来指定页表的起始物理地址，页表中的页表项用来指定物理页的起始地址，这样就构成了一个二级页表映射。页目录的起始物理地址由x86处理器中的控制寄存器CR3指定。

按照下面的步骤将线性地址映射为物理地址：

1. 首先需要算出线性地址中的页目录号、页表号和页内偏移，它们分别对应了32位线性地址的10位+10位+12位，所以0x10003004的页目录号为64，页表号为3，页内偏移为4。
2. 页目录表的起始物理地址由CR3寄存器指定。在Bochs的命令窗口中输入命令“creg”

可以查看CR3寄存器的值，如下：

```
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page fault laddr=0x0000000010002fac
CR3=0x00000000
PCD=page-level cache disable=0
PWT=page-level writes transparent=0
CR4=0x00000000: osxmmexcpt osfxsr pce pge mce pae pse de tsd pvi vme
EFER=0x00000000: ffxsr nxe lma lme sce
```

图8-15 控制寄存器的值

CR3寄存器的值为0，说明页目录的起始物理地址为0。

3. 在Bochs的命令窗口中输入命令“xp /w 0+64*4”，在页目录中查看页目录号为64的页表项，如下图（如果读者得到页目录号与这里的不一致，请读者使用自己得到的值进行后续的计算）：

```
0x00000000000000100 <bogus+ 0>: 0x00fa7027
```

图8-16 页目录号为64的页表项

其中027是属性位的值，请读者自己分析这些属性值的意义。页表所在物理页框号为0x00fa7，即页表的起始物理地址为0x00fa7000。

4. 在Bochs的命令窗口中输入命令“xp /w 0x00fa7000+3*4”，在页表中查看页表号为3的页表项，如下图：

```
0x000000000000fa700c <bogus+ 0>: 0x00fa2067
```

图8-17 页表号为3的页表项

其中0x00fa2是物理页所在的页框号，所以物理页的起始地址为0x00fa2000。将物理页的起始地址与页内偏移4加在一起得到0x00fa2004，这就是变量i的物理地址了。

5. 在Bochs的命令窗口中输入命令“xp /w 0x00fa2004”，查看从该物理地址开始的4个字节的值，也就是变量i的值，如下图：

```
0x000000000000fa2004 <bogus+ 0>: 0x12345678
```

图8-18 变量i的值

可以验证这个值与程序中变量i的初值是一样的。

6. 在Bochs的命令窗口中输入命令“setpmem 0x00fa2004 4 0”，将从物理地址0x00fa2004开始的4个字节的值都设为0。然后再使用命令“c”让Bochs继续运行，可以看到应用程序退出了，说明变量i的值在被修改为0后结束了死循环。

之前调试了全局变量的地址映射过程，下面请读者自己调试局部变量的地址映射过程。可以将loop应用程序的源代码文件修改为如下的代码，通过物理内存修改局部变量i的值使应用程序结束。

```
#include <stdio.h>
int main(void)
{
    int i = 0x12345678;
    printf("The logical/virtual address of i is 0x%08x\n", &i);
    fflush(stdout);
    while(i)
    ;
    return 0;
}
```

3.4 查看应用程序进程的页目录和页表

之前的练习已经让读者详细了解了逻辑地址、线性地址和物理地址的映射过程。其中逻辑地址到线性地址的映射过程涉及到了存储器的分段管理，这部分只要知道了段基址和段限长等概念就可以了。线性地址到物理地址的映射过程涉及到了存储器的二级页表管理，为了读者能够对二级页表有一个直观的、整体上的认识，并为后面通过共享物理页的方法来实现内存共享功能，这里使用一个应用程序将进程的二级页表打印出来，供读者仔细研究。请按照下面的步骤进行实验：

1. 新建一个Linux011 Kernel项目。
2. 添加一个系统调用号为87的系统调用（添加系统调用的方法请参考实验四），该系统调用的内核函数sys_table_mapping可以写在kernel/sys.c文件的末尾。其源代码可以参见“学生包”中本实验对应文件夹下的table.c文件。
3. 在sys_table_mapping函数中调用了名称为fprintk的函数用于向标准输出打印信息，而没有使用printk函数，这是由于本实验需要输出的内容较多，需要输出到文件中以便查看，但是printk函数虽然可以在屏幕上进行输出，但是却不能输出到文件中，所以需要实现一个fprintk函数。该函数可以在kernel/printk.c文件中实现，其源代码参见“学生包”中本实验对应文件夹下的fprintk.c文件。还需要在include/linux/kernel.h文件中添加该函数的声明。该函数的第一个参数fd是文件描述符，其值为1时将输出的信息写入标准输出stdout。
4. 在sys_table_mapping函数的开始位置还调用了calc_mem函数，此函数会计算内存中空闲页面的数量以及各个页表中映射的物理页的数量并显示。在此调用此函数是为了验证sys_table_mapping函数输出的页目录和页表的数量是否正确。由于函数calc_mem中使用的是printk函数，所以还无法将输出内容保存到文件中，请读者将其替换为fprintk函数。
5. 源代码修改完毕后按F7生成项目，确保没有语法错误和警告。
6. 按F5进行调试，待Linux011完全启动后，使用vi编辑器新建一个main.c文件。编辑main.c文件中的源代码如下：

```
#define __LIBRARY__
#include <unistd.h>
#define __NR_table_mapping 87
_syscall0(int, table_mapping)
int main()
{
    table_mapping();
    return 0;
}
```

7. 保存main.c文件后退出vi编辑器，依次执行如下命令：

```
gcc main.c -o table
sync
table > a.txt
mcopy a.txt b:a.txt
```

8. 结束调试后，在“项目管理器”窗口中双击floppyb.img文件，使用软盘编辑器工具打开。将其中的a.txt文件复制到Windows的某个文件夹中。
9. 将Windows文件夹中的a.txt文件拖动到Linux lab中释放，分析输出的结果。

在sys_table_mapping函数中有一行代码 `page_table_base = 0xFFFFF000 & entry`，是将一个页表的物理地址转换为它的逻辑地址，其中entry是页目录项，取其中的高20位就可以得到页表的物理地址。为什么页表的物理地址和它的逻辑地址相同呢？下面结合文件a.txt中的输出结果进行说明。文件a.txt的开始部分的数据如下：

PDE: 0x0 (0x0)→0x1

PTE: 0x0 (0x0)→0x0

PTE: 0x1 (0x1000)→0x1

PTE: 0x2 (0x2000)→0x2

PTE: 0x3 (0x3000)→0x3

PTE: 0x4 (0x4000)→0x4

一个已知的前提是页目录的物理页框号是0x0，并且页目录的第0项指向物理页框号为0x1的页表，而此页表的第0项又指回了物理页框号为0x0的页目录，页表的第1项指向物理页框号为0x1的页表，也就是它自己，并依此类推，即第一个页表中的所有页表项指向了1024个页表（包括页目录），如图8-19所示。正是由于这样的布局和映射关系，确保了页表的物理地址与其线性地址相同，而且内核态中段基址为0，所以页表的物理地址也就与其逻辑地址相同了。例如，逻辑地址0x1000，其线性地址仍然是0x1000，在分页变换时，其高10位为0，所以在页目录中取偏移为0的页目录项，该页目录项对应的页表的物理页框号为0x1，物理地址即为0x1000，线性地址中间10位为1，所以在页表中取偏移为1的页表项，该页表项对应的物理页的页框号为0x1，物理地址仍然为0x1000，可以看到，页表的逻辑地址与物理地址相同。请读者用逻辑地址0x2000计算一下其对应的物理地址，看看物理页框号为0x2的页表，它的物理地址与逻辑地址是否相同。

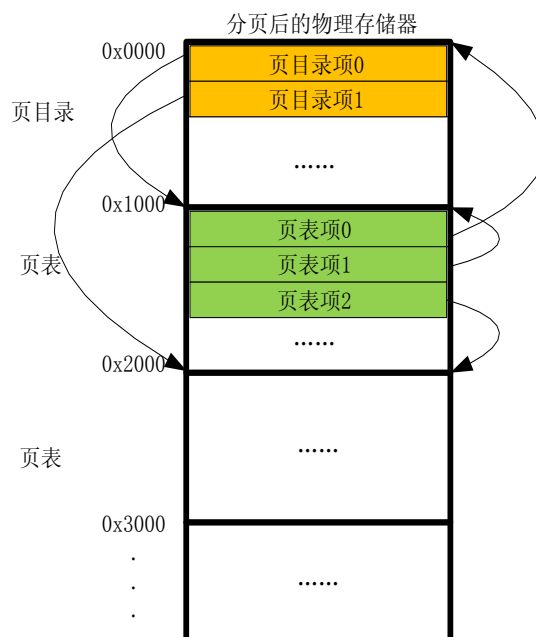


图8-19 页目录和页表在物理内存的布局示意图

3.5 用共享内存做缓冲区解决生产者——消费者问题

读者通过前面的实验内容学习了Linux 0.11管理物理内存的方式，以及使用x86处理器提供的分段功能和二级页表功能实现地址映射的过程。在此基础上，读者可以将一个共享的物理页映射到不同的逻辑地址空间，从而实现在进程间共享内存的方法，进而使用共享内存作为缓冲区来解决生产者——消费者问题。

本实验需要读者在Linux 0.11的内核中添加shmget与shmat两个共享内存的系统调用函数。在“学生包”本实验对应的文件夹下提供了一个sem.c文件，其中包含了信号量的四个系统调用函数（与实验七中的相同），以及共享内存的两个系统调用函数所对应的内核函数sys_shmget和sys_shmat，这里只是实现了一个简化后的版本：只能将一个物理页映射到不同进程的逻辑地址空间的末尾。请读者仔细阅读其中的源代码，进而理解下面的内容。

系统调用函数shmget的作用是新建或者打开一页共享内存，并将该共享内存的ID返回，其内核函数sys_shmget的原型定义如下：

```
int sys_shmget(int key, int size)
```

参数key用来指定标识共享内存的键值，结合源代码可知key就是vector数组的下标，所以其必须是一个小于数组长度的正整数；参数size用来指定共享内存的大小，由于总是调用get_free_page函数分配一个物理页作为共享内存，所以此参数只是用来进行简单的校验。在函数的最后将物理页的基地址放入由下标key指定的数组项中，并返回物理地址作为ID。

系统调用函数shmat的作用是将由ID指定的共享内存加入到当前进程的二级页表映射中，并返回其逻辑地址，其内核函数sys_shmat的原型定义如下：

```
void* sys_shmat(int shmid, const void *shmaddr)
```

参数shmid用来指定共享内存的ID，即由shmget函数返回的ID，其实质是共享页的物理地址；参数shmaddr在这里没有用到。在这个函数中主要是调用了put_page函数将物理页加入二级页表映射。put_page函数的第一个参数是物理页的基地址，第二个参数是物理页需要映射到的线性地址。为了简单，这里直接将共享的物理页映射到了应用程序的末尾，所以传入的第二个参数是current->start_code + current->brk。其中current->start_code是应用程序代码的起始线性地址，current->brk是应用程序所占内存的长度。由于sys_shmat函数需要返回共享内存的逻辑地址，而且应用程序在用户态时，其段基址与代码的起始地址相同，即current->start_code在应用程序空间的逻辑地址为0，所以current->brk就是需要返回的逻辑地址。

请读者按照下面的步骤完成本实验：

1. 新建一个Linux011 Kernel项目。
2. 将sem.c文件中的四个信号量的系统调用和两个共享内存的系统调用添加到内核中。添加系统调用的方法可以参考实验四和实验七，具体步骤这里不再详细说明。
3. 按F7生成项目，确保没有语法错误和警告。
4. 新建一个Linux011应用程序项目，在windows资源管理器中将前面Linux011 Kernel项目文件夹下floppya.img文件拷贝覆盖新建的应用程序项目文件夹下的floppya.img，这样就可以在应用程序中使用已经添加的信号量和共享内存的系统调用了。
5. 将“学生包”本实验文件夹下的pc.c文件拖放到Linux Lab中打开，用其中的源代码替换刚创建的应用程序项目中的LinuxApp.c中的源代码。这些源代码仍然是使用文件作为生产者和消费者之间的共享缓冲区，请读者在此基础上将其修改为使用共享内存作为缓冲区。

提示：

- 在应用程序中定义共享内存系统调用函数的调用号和函数时，可以参考下面的代码：

```
#define __NR_shmget 91
#define __NR_shmat 92
```

```
_syscall2(int, shmget, int, key, int, size)
_syscall2(int, shmat, int, shmid, const void*, shmaddr)
```

- 在生产者进程和消费者进程中都需要分别调用 shmget 函数和 shmat 函数来得到共享内存的起始地址，可以参考下面的代码：

```
shmid = shmget(key, 4096);
startaddress = shmat(shmid, NULL);
```

6. 代码修改完毕后，按 F7 生成应用程序项目，按 F5 启动调试。
7. 依次执行下面的命令：

```
mcopy b:linuxapp.exe app
chmod +x app
app
```

应确保程序执行的结果与实验七中的图 7-1 显示的结果一致。

注意：此应用程序执行到最后可能会出现错误提示“trying to free free page”，这是由于生产者进程和消费者进程在退出时都会释放共享的物理页，导致一个物理页被回收两次。为了避免操作系统提示此错误信息并中断执行，可以暂时把文件 mm/memory.c 中 free_page 函数的最后一行（第 141 行）代码注释掉。

四、思考与练习

1. 请读者认真体会本实验中关于地址映射的内容，尝试列出映射过程中最为重要的几步（不超过四步），并给出获得的实验数据。
2. 在本实验第3.2节的loop程序退出后，如果读者接着再运行一次，并再次进行地址跟踪，会发现有哪些异同？尝试说明原因？
3. 参考下面的代码，在Linux 0.11内核中写一个系统调用函数，并在Linux 0.11的应用程序中调用此函数，然后仿照本实验第3.2节的内容跟踪变量i的逻辑地址、线性地址、物理地址的映射过程，最后通过将物理内存中变量i的值修改为0的方式使应用程序退出。

```
volatile int i = 0x12345678;
void sys_testg()
{
    printk("The logical/virtual address of i is 0x%08x\n", &i);
    while(i)
        ;
}
```

定义变量 i 时使用“volatile”关键字目的是防止编译器对其进行优化，否则 i 的值会被拷贝到寄存器中，而通过Bochs调试命令修改的是内存中 i 的值，不是寄存器中的值，会导致程序无法退出死循环。

4. 将上一个练习中的变量 i 修改为 sys_testg 函数内的局部变量，仍然通过将物理内存中变量 i 的值修改为 0 的方式使应用程序退出。请读者结合 Linux 0.11 内核的逻辑地址空间的内存布局，体会内核中的全局变量与局部变量在逻辑地址空间中的位置有什么不同。
5. 将本实验3.5中的Linux 0.11应用程序修改为一个生产者与多个消费者使用共享内存作为缓冲区的情况。（提示：关键是需要将多个消费者从缓冲池中取产品的位置放入共享内存中的适当位置。）
6. 在本实验3.5中遇到了共享的物理页被多个进程重复释放，导致操作系统报告错误并中断运行的问题，暂时是通过注释掉文件mem/memory.c中free_page函数的最后一行（第

141行)代码来解决的。请读者实现一个简化版本的关闭共享内存的系统调用函数shmdt, 其函数原型可以为:

```
void shmdt(int key, const void* startaddr)
```

参数key是共享内存的键值, 即为共享内存数组的下标; 参数startaddr是共享内存的逻辑地址。在每个生产者进程和消费者进程退出前, 都必须调用此函数关闭共享内存, 最终确保无论是一个生产者与一个消费者同步运行, 还是一个生产者与多个消费者同步运行都不会再出现应用程序退出时操作系统报告错误的情况。

提示:

1. 读者需要实现一个将共享的物理页从当前进程的二级页表映射中移除的内核函数, 其原型可以为:

```
void cancel_mapping(const void* linearaddr)
```

参数linearaddr是需要取消映射的物理页的线性地址, 根据此线性地址找到对应的页表项, 然后将页表项置为0即可。

2. 之前的源代码中只是用全局的vector数组保存了共享内存的物理页的基址, 现在需要为共享内存定义一个结构体, 其中除了保存物理页的基址外, 还需要保存共享内存的引用计数 (可以参考信号量的引用计数), 再使用此结构体定义一个全局的共享内存数组。当进程调用shmdt函数关闭共享内存时, 首先将共享内存的引用计数减1, 然后调用cancel_mapping函数将共享的物理页从二级页表映射中移除。由于shmdt的第二个参数startaddr是共享内存的逻辑地址, 需要为其加上current->start_code转换为线性地址后, 才能作为cancel_mapping函数的参数。最后, 判断引用计数的值如果大于0, 说明仍然有其他进程在使用此共享内存, 就结束 (不释放物理页); 否则, 需要调用free_page函数释放物理页。

实验九 页面置换算法与动态内存分配

实验性质：验证、设计

建议学时：2 学时

实验难度：★★★★☆

一、 实验目的

- 掌握OPT、FIFO、LRU、LFU、Clock等页面置换算法；
- 掌握可用空间表及分配方法；
- 掌握边界标识法以及伙伴系统的内存分配方法和回收方法。

二、 预备知识

页面置换算法

请求分页虚存管理的实现原理是：把作业的所有分页副本存放在磁盘中，当它被调度投入运行时，首先把当前需要的页面装入内存，之后根据程序运行的需要，动态装入其他页面；当内存空间已满，而又需要装入新页面时，根据某种算法淘汰某个页面，以便装入新页面。因此，在页表中必须说明哪些页已在内存，存在什么位置；哪些页不再内存，它们的副本在磁盘中的什么位置。还可以设置页面是否被修改过，是否被访问过，是否被锁住等标志供淘汰页面使用。

在地址映射过程中，若页表中发现所要访问的页不在内存，则产生缺页异常，操作系统接到此信号后，就调出缺页异常处理程序，根据页表中给出的磁盘地址，将该页面调入内存，使作业继续运行下去。如果内存中有空闲页，则分配一个页，将新调入页面装入，并修改页表中相应页表项的驻留位以及相应的内存块号；若此时内存中没有空闲页，则要淘汰某页面，若该页在此期间被修改过，还要将其写回磁盘，这个过程称为页面替换。

动态内存分配

边界标识法是操作系统中用以进行动态分区分配的一种存储管理方法。系统将所有的空闲块链接在一个双重循环链表结构的可利用空间中；分配可按首次拟合进行，也可按最佳拟合进行。其特点在于：在每个内存区的头部和尾部两个边界上分别设有标识，以标识该区域为占用块或空闲块，使得在回收用户释放的空闲块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便将所有地址连续的空闲块组合成一个尽可能大的空闲块。

伙伴系统是操作系统中用到的另一种动态存储管理办法。它和边界标识法类似，在用户提出申请时，分配一块大小“恰当”的内存区给用户；反之，在用户释放内存时即回收。所不同的是：在伙伴系统中，无论是占用块或空闲块，其大小均为 2 的 k 次幂（k 为某个整数）。

三、 实验内容

由于在 Linux 0.11 内核中还没有实现虚拟内存的管理，当然也不存在页面置换算法，而且修改 Linux 0.11 内核的动态内存分配会比较复杂，所以本实验在 Windows 控制台中通过模拟来实现页面置换算法和动态内存分配。

3.1 页面置换算法

3.1.1 准备实验

1. 启动 Engintime Linux Lab。
2. 使用“控制台应用程序(c)”模板，新建一个 Windows 控制台应用程序项目。

3.1.2 查看最佳页面置换算法(OPT)和先进先出页面置换算法(FIFO)的执行过程

在“学生包”本实验对应文件夹中，提供了实现最佳页面置换算法（OPT）和先进先出页面置换算法（FIFO）的源代码文件 page.c。使用 Linux Lab 打开此文件（将文件拖到 Linux Lab 窗口中释放即可打开），仔细阅读此文件中的源代码和注释，查看 OPT 和 FIFO 是如何实现的。

其中，在 main 函数中定义了一个数组 PageNumofR[] 来存放页面号引用串，定义了一个指针*BlockofMemory 指向一块存放页面号的内存块。依次将数组 PageNumofR[] 中的页面装入内存，根据不同的页面置换算法淘汰内存中的页面。

按照下面的步骤查看 OPT 和 FIFO 的执行过程：

1. 使用 page.c 文件中的源代码，替换之前创建的 console.c 文件内的源代码。
2. 按 F7 键生成修改后的 c 控制台应用程序项目，确保没有语法上的错误。
3. 按 Ctrl+F5 执行此程序，查看 OPT 和 FIFO 的执行过程。执行结果如下图所示：

```

*****最佳页面置换算法:*****
页面引用串是: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2
内存中页块为: 7 0 0 0 0
内存中页块为: 7 0 0 0 0
内存中页块为: 7 0 1 0 0
内存中页块为: 7 0 1 2 0
内存中页块为: 7 0 1 2 0
内存中页块为: 3 0 1 2 0 淘汰页面号为: 7
内存中页块为: 3 0 1 2 0
内存中页块为: 3 0 4 2 0 淘汰页面号为: 1
内存中页块为: 3 0 4 2 0
内存中页块为: 3 0 4 2 0
内存中页块为: 3 0 4 2 0
内存中页块为: 3 0 4 2 0
内存中页块为: 3 0 4 2 0
内存中页块为: 1 0 4 2 0 淘汰页面号为: 3
内存中页块为: 1 0 4 2 0
OPT缺页率为: 0.375
抖动的次数为: 3
*****先进先出页面置换算法:*****
页面引用串是: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2
内存中页块为: 7 0 0 0 0
内存中页块为: 7 0 0 0 0
内存中页块为: 7 0 1 0 0
内存中页块为: 7 0 1 2 0
内存中页块为: 7 0 1 2 0
内存中页块为: 3 0 1 2 0 淘汰页面号为: 7
内存中页块为: 3 0 1 2 0
内存中页块为: 3 4 1 2 0 淘汰页面号为: 0
内存中页块为: 3 4 1 2 0
内存中页块为: 3 4 1 2 0
内存中页块为: 3 4 1 2 0
内存中页块为: 3 4 1 2 0
内存中页块为: 3 4 1 2 0
内存中页块为: 3 4 1 2 0
内存中页块为: 3 4 1 2 0
FIFO缺页率为: 0.29
抖动的次数为: 2
请按任意键继续. . .

```

图 9-1 OPT 和 FIFO 的执行过程

3.1.3 完成最近最久未使用页面置换算法和简单的 clock 页面置换算法

“学生包”本实验对应文件夹下的 page2.c 文件中提供了两个算法的参考代码。将 page2.c 文件内的源代码拷贝到 console.c 中，仔细阅读其中的源代码，并根据提示实现最近最久未使用(LRU)页面置换算法和简单的 Clock 页面置换算法。

有兴趣的读者还可以使尝试写出最少使用(LFU)页面置换算法和页面缓冲页面置换(PBA)算法。

3.2 动态内存分配

3.2.1 准备实验

使用“控制台应用程序(c)”模板，新建一个 Windows 控制台应用程序项目。

3.2.2 边界标识法的设计实现

在“学生包”本实验对应文件夹中，提供了实现边界标识法的源代码文件 btm.c，将其拖到 Linux Lab 中释放，即可打开此文件。仔细阅读其中的源代码及注释，阅读时着重注意以下几点：

- 定义 MINISIZE 的目的是为了防止在多次分配以后链表中出现一些极小的、总也分配不出去的空闲块；
- 表结构体


```
struct table{
    unsigned long address;
    unsigned long length;
    int flag;
};
```

 其中的 address 是内存的起始地址；length 是内存长度，单位为字节；flag 是标志位，为 0 表示空栏目，为 1 表示内存未使用。
- 定义了 used_table[] 和 free_table[] 两个数组来分别存放已分配区表和空闲区表；
- allocate 函数是采用首次适应算法分配内存的。
- 回收内存 reclaim 函数，在回收内存时要跟空闲分区中的左右进行比较，若有空闲块则进行内存合并。
- SetColor 是设置字体的函数，是为了使输出结果更加醒目。

按照下面的步骤查看边界标识法的实现过程：

1. 用 btm.c 文件内的代码替换刚刚创建的控制台应用程序项目的 console.c 文件内的代码。
2. 按 F7 生成项目，确保没有语法错误。
3. 按 Ctrl + F5 运行项目，查看运行结果。其结果如下图所示：



```
选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:
```

图 9-2 边界标识法的运行结果

系统会停留在此界面等待用户输入功能项序号 0-3 中的其中一个。系统会根据用户输入的序号执行相应的功能。如果没有按数字 0，系统会执行完相应的功能后继续停留在此界面等待用户的输入，直到用户按 0 退出应用程序。

例如，输入 1 来分配内存，所需内存长度 102350，由于初始化的内存大小为 102400 字，申请内存以后系统剩余可用内存大小小于 100 字，所以此时再输入 3 查看内存，可以看到系统将 102400 字的内存全都分配给来该作业。如图 9-3 所示。如果输入的作业长度小于 102300，则申请多少内存，系统就分配多少内存。如果申请的作业数大于 10 个，则会提示“已分区

表已满，无法继续分配内存”。如图 9-4 所示。当然还可以输入 2 来回收内存。如果要回收的内存左右有空闲区，就会与之合并留在可利用空间表，否则就作为一个结点插入到可利用空间表中。如图 9-5 所示。

分配空间时，采用的是首次适应算法，请读者尝试采用循环首次适应算法和最佳适应算法分别实现边界标识法。

```

选择功能项: <0-退出 1-分配内存 2-回收内存 3-显示内存>: 1
所需内存长度: 102350
选择功能项: <0-退出 1-分配内存 2-回收内存 3-显示内存>: 3
空闲区表:
起始地址 分区长度 标志
10240 102400 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
输入任意字符输出已分配表...
已分配区表:
起始地址 分区长度
10240 102400
0 0
0 0

```

图 9-3 申请 102350 字节后的内存分配

```

选择功能项: <0-退出 1-分配内存 2-回收内存 3-显示内存>: 1
所需内存长度: 120
已分区表已满, 无法继续分配内存!

```

图 9-4 申请第 11 个内存块时的内存显示

```

选择功能项: <0-退出 1-分配内存 2-回收内存 3-显示内存>: 2
输入要回收分区的首地址: 107294
选择功能项: <0-退出 1-分配内存 2-回收内存 3-显示内存>: 3
空闲区表:
起始地址 分区长度 标志
10240 96554 1
107294 4090 1
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
输入任意字符输出已分配表...
已分配区表:
起始地址 分区长度
111640 1000
111384 256
0 0
106794 500
0 0
0 0

```

图 9-5 回收内存

3.2.3 伙伴系统的设计实现

在“学生包”本实验对应文件夹中，提供了实现伙伴系统的部分源代码文件 `buddy.c`。将其拖到 Linux Lab 中释放，即可打开此文件。用此文件内的代码替换刚刚创建的 Windows 控制台应用程序项目的 `console.c` 文件内的代码。仔细查看其中的源代码及注释，并根据注释设计完善伙伴系统。

阅读代码时注意以下几点：

- 伙伴系统可利用空间表的结构体

```

typedef struct _Node{
    struct _Node *llink;    //指向前驱节点
    int bflag;              //块标志, 0: 空闲, 1: 占用
    int bsize;              //块大小, 值为 2 的幂次 k
    struct _Node *rlink;    //指向后继节点
}Node;

```

可利用空间表的头结点的结构体

```

typedef struct HeadNode{
    int nodesize;          //链表的空闲块大小
    Node *first;           //链表的表头指针
}FreeList[M+1];

```

- 在 `AllocBuddy` 函数中分配空闲块以后，还需要将剩余块插入相应的子表中。
- `Reclaim` 函数回收内存时，需要先查找其伙伴是否为空闲块。若否，则只要将释放的空闲块简单插入相应的子表中即可；若是，则需在相应子表中找到其伙伴并删除之，然后在判别合并后的空闲块的伙伴是否是空闲块。依此重复，直到归并所得空闲块的伙伴不是空闲块时，再插入到相应的子表中去。

执行程序的输出结果如图 9-6 所示。例如，输入 1 选择分配内存，然后输入作业所需长度 20，执行完以后再输入 3 查看内存分布情况。由于 $16 < 20 < 32$ ，所以系统会分配给该作业 32 字的内存块，如图 9-7 所示。需要注意的是，由于每个字由两个字节构成，一个字节占用 8 位内存，所以相邻块之间首地址的差值为块的大小与 16 的乘积。

选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:

图 9-6 伙伴系统运行的结果

```

选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:1
输入所需内存空间长度：20
选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:3
可利用空间：
    块的大小    块的首地址    块标志<0:空闲 1:占用>
      32        4079600        0
      64        4080112        0
     128        4081136        0
     256        4083184        0
     512        4087280        0

已利用空间：
    占用块号  占用块的首地址  块大小  块标志<0:空闲 1:占用>
        0        4079088        32        1
  
```

图 9-7 申请 20 字内存块后的内存分布情况

四、思考与练习

- 1、新建一个 linux011 kernel 的项目，打开 lib/malloc.c 文件，仔细阅读其中的源代码及注释，查看一下 linux 0.11 系统动态分配和回收内存的方法。系统使用的方法是与本实验使用的边界标识法和伙伴系统法不同。请读者尝试分别用边界标识法和伙伴系统算法来修改 malloc 函数和 free 函数来修改系统的动态分配内存和回收内存的方法。
- 2、x86 平台为实现改进型的 Clock 页面置换算法提供了硬件支持，它实现了二级映射机制，它定义的页表项的格式如下：

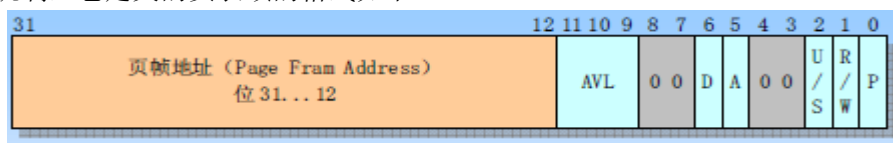


图9-8 页目录和页表中的表项的格式

其中位 5 是访问标志，当处理器访问页表项映射的页面时，页表表项的这个标志就会被置为 1。位 6 是页面修改标志，当处理器对一个页面执行写操作时，该项就会被置为 1。

在 Linux 0.11 的内核中写一个系统调用函数来实现改进型的 Clock 页面置换算法的模拟。参考实验八的第 3.4 节，在此函数中将二级映射表打印出来，然后实现一个算法来根据页表项中的访问位和修改位计算出应该置换出的页面。在此函数中不必完全实现 Clock 页面置换算法，只要能判断出应该置换的页面即可。再编写一个应用程序，在应用程序中调用此系统调用函数。

实验十 字符显示的控制

实验性质：验证

建议学时：2 学时

实验难度：★★★★☆☆

一、实验目的

- 加深对操作系统设备管理基本原理的认识，掌握键盘中断、扫描码等概念；
- 掌握 Linux 对键盘设备和显示器终端的处理过程。

二、预备知识

操作系统管理的设备可以按照信息交换的单位分为字符设备（Character Device）和块设备（Block Device）。字符设备以字节为单位，而块设备以块为单位（块大小通常为512或1024字节）。终端是一种典型的字符设备，它具有多种类型，通常使用tty来简称各种类型的终端设备。tty是Teletype的缩写，Teletype是一种由Teletype公司生产的最早出现的终端设备，外观很像电传打字机。在Linux 0.11内核中使用tty_struct结构体定义终端设备，主要用来保存终端设备当前的参数设置、所属的前台进程组ID和字符ID缓冲队列等信息。结构体tty_struct在include/linux/tty.h文件的第65行定义如下：

```
struct tty_struct{
    struct termios termios;
    int pgrp;
    int stopped;
    void (*write)(struct tty_struct *tty);
    struct tty_queue read_q;
    struct tty_queue write_q;
    struct tty_queue secondary;
};
```

其中的三个队列（典型的环形缓冲区）是本实验最关心的。读缓冲队列read_q用于临时存放从键盘或串行终端输入的原始字符序列；写缓冲队列write_q用于存放写到显示设备或串行设备去的数据；根据辅助队列secondary用于存放从read_q中取出的经过行规则程序处理（过滤）过的数据。

Linux 0.11内核使用tty_struct结构体定义了一个数组tty_table[]用来保存操作系统中每个终端设备的信息。通常支持三个终端，其中数组的第0项是控制台终端tty0，包括显示设备和键盘设备；接下来的两项是串行设备终端tty1和tty2。

在init/main.c文件的第228行有以下几行源代码：

```
(void)open( "/dev/tty0", 0_RDWR, 0); //用读写访问方式打开设备 "/dev/tty0",
//相当于stdin标准输入设备
(void)dup(0); //复制句柄，产生句柄1号——stdout标准输出设备
(void)dup(0); //复制句柄，产生句柄2号——stderr标准出错输出设备
```

可以看到，在Linux 0.11应用程序中使用的标准输入stdin（值为0）、标准输出stdout（值为1）和标准错误stderr（值为2）都是终端设备tty0的句柄。

详细内容请读者阅读《Linux内核完全注释》第4.6节，第5.4节以及第10章的内容，附

录2提供了ASCII码表，附录5提供了第1套键盘扫描码。

三、实验内容

本实验首先引导读者调试Linux 0.11内核中键盘中断服务程序的执行过程和显示字符的方式，从多个层面上帮助读者理解终端设备tty0的工作原理。然后修改Linux 0.11的键盘设备和显示设备处理代码，对键盘输入和字符显示进行非常规的控制：在Linux 0.11刚启动时，一切如常；当按一次F12后，向终端输出的所有字母和数字都会被替换为“*”字符；再按一次F12后，又恢复正常；第三次按F12后，再进行替换，依此类推。通过这个替换字符的实验，读者可以对Linux 0.11通过tty0终端对键盘设备和显示设备的管理方式有一个更直观的认识。最后，在之前实验的基础上，继续实现一个贪吃蛇游戏。

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。

3.2 调试键盘中断服务程序的执行过程和显示字符的方式

键盘设备是一种典型的中断驱动的设备，在kernel/chr_drv/console.c文件中的第843行定义了一个con_init函数，负责为键盘中断设置一个中断服务程序，其源代码如下：

```
void con_init(void)
{
    set_trap_gate(0x21, &keyboard_interrupt); //设置键盘中断陷阱门
}
```

请读者自行查找一下这个con_init函数是如何在Linux 0.11内核的初始化过程中被调用的。

每当键盘设备上的一个键被按下或者弹起时就会触发一次键盘中断，键盘中断服务程序keyboard_interrupt就会被调用。keyboard_interrupt函数是一个汇编函数，源代码文件kernel/chr_drv/keyboard.s的第73行是此函数的入口点。当此函数被调用时，寄存器EAX中保存了键盘的扫描码。

读者可以按照下面的步骤调试键盘中断服务程序的执行过程：

1. 在kernel/chr_drv/keyboard.s文件的第89行和第109行分别添加一个断点。第89行是响应键盘中断并调用按键处理程序的位置，通常情况下，按键处理程序会将键盘扫描码转换为使用ASCII码表示的字符，然后将字符放入tty0设备的读队列(read_q)中。第109行调用do_tty_interrupt函数将tty0设备读队列(read_q)中的字符复制成规范模式字符，然后存放在tty0设备的辅助队列(secondary)中。
2. 按F7生成项目，然后按F5启动调试。
3. 待Linux 0.11完全启动后，按键盘上的键A，将在文件kernel/chr_drv/keyboard.s的第89行处中断。此时，将鼠标移动到源代码中eax寄存器名称上，会显示eax寄存器的值为0x1E，即为按键A的接通码。
4. 按F11单步调试，会进入由key_table表中第0x1E项所指向的键盘处理函数do_self。
5. 按F10单步调试，直到在第384行停止。第381和第383行会使用键盘扫描码0x1E作为key_map映射表的索引值，取对应的ASCII码，并放入al寄存器中。此时，将鼠标移动到源代码中eax寄存器名称上，会显示eax寄存器的值为0x61，即为字符“a”的ASCII码。
6. 按F10单步调试，直到在第412行停止。此行的put_queue函数会将al寄存器中的字符放入tty0设备的读队列(read_q)中。
7. 按F5继续调试，会在keyboard.s文件的第109行的断点处中断，准备调用

do_tty_interrupt函数。

8. 按F11进入do_tty_interrupt函数。可以看到该函数只是调用了copy_to_cooked函数。此时，将鼠标移动到函数参数tty上，会显示参数tty的值为0x0，说明指定的是序号为0的tty设备。
9. 按F11进入copy_to_cooked函数。在“快速监视”对话框中查看表达式“*tty”的值，可以看到tty0设备的三个队列中的数据。请读者自己分析一下这些数据。
10. 按F10单步调试，直到在第199行停止。其中第195行是从tty0的读队列队尾获取一个字符到变量c。此时，将鼠标移动到源代码中变量c的名称上，会显示变量c的值为0x61，即为字符“a”的ASCII码。
11. 按F10单步调试，直到在第303行停止。在此调试过程中有若干个if语句，由于字符“a”统统不满足这些判断情况，所以会直接跳过这些语句。第300行的PUTCH函数会将字符“a”放入到tty0的辅助队列（secondary）中，当所有字符都被放入到tty的辅助队列中之后就会跳出循环。最后，在第303行调用wake_up函数唤醒等待该辅助队列的进程，使其处理辅助队列中的字符。使用“快速监视”对话框查看表达式“*tty->secondary.proc_list”的内容，可以看到阻塞在辅助队列中的进程的pid为4，说明其为Shell进程，可以理解为Shell进程从标准输入stdin中获取输入数据时发生了阻塞，当有输入数据后就会被唤醒。
12. 继续按F10单步调试，直到在kernel/chr_drv/keyboard.s文件的第117行停止。至此，键盘中断服务程序就执行完毕了，会调用iret指令从中断服务程序返回，然后执行进程调度，让刚刚被唤醒的Shell进程继续执行。

Shell 进程被唤醒后会继续执行。虽然这里没有提供 Shell 程序的源代码，但是 Shell 程序的基本流程还是比较简单的，如下：

- 1) 首先，Shell 进程调用 read 函数读取标准输入 stdin（也可以调用 C 标准库的 getchar 函数）。在用户还没有输入字符的情况下，Shell 进程会阻塞在 tty0 的辅助队列上。
- 2) 当用户从键盘输入一个字符后，tty0 的辅助队列中就有字符了（也就是之前调试的过程），此时 Shell 进程会被唤醒，read 函数在读取到用户输入的字符后就可以返回了，其本质可以理解为将 tty0 设备的辅助队列中的字符读取到 Shell 进程在用户空间的缓冲区中。
- 3) 然后，Shell 进程调用 write 函数写标准输出 stdout（也可以调用 C 标准库的 printf 函数），将用户空间缓冲区中的字符输出到显示设备，其本质可以理解为将用户空间缓冲区中的字符写入到 tty0 设备的写队列（write_q）中，同时将写队列中的字符输出到显示设备。
- 4) 最后，Shell 进程还需要对获取到的字符进行必要的处理，例如，当获取到的是回车字符时，需要将缓冲区中的内容（可能是“ls”）作为一个命令来执行，待命令执行完毕后再打印命令提示符，然后回到步骤 1，继续等待用户输入。

请读者按照下面的步骤调试 Shell 进程被唤醒后从 tty0 的辅助队列读取字符，然后再将字符写入 tty0 的写队列，最终将字符输出到显示设备的过程：

1. 在/kernel/char_drv/tty_io.c 文件中 tty_read 函数的第 358 行添加一个断点。
2. 按 F5 继续运行，会在刚刚添加的断点处中断。Shell 进程就是在执行第 357 行代码时阻塞在 tty0 的辅助队列的，现在由于键盘中断处理程序向辅助队列中放入了数据并唤醒了 Shell 进程，使其从第 357 行的 sleep_if_empty 函数中返回，所以，

- Shell 进程就会在刚刚添加的断点处中断。将鼠标放到 `tty_read` 函数的第一个参数上（第 309 行），显示其值为 0，在第 321 行根据此参数的值获取到了 `tty0`。
3. 选择“调试”菜单“窗口”中的“调用堆栈”，打开调用堆栈窗口，双击其中的 `sys_read` 函数调用帧，可以切换到 `sys_read` 函数中。此时，将鼠标移动到 `sys_read` 函数的第一个参数 `fd` 上，显示其值为 0，说明 Shell 程序确实调用了 `read` 函数（或 `getchar` 函数），并且传入的文件句柄为 0，也就是 `stdin`。
 4. 按 F10 单步调试，直到在第 372 行停止。可以看到第 363 行将辅助队列中的字符放入了变量 `c` 中，也就是之前输入的字符“a”。第 372 行将该字符放入用户数据段的缓冲区 `buf` 中。Shell 程序就可以在用户空间处理 `buf` 中的字符了。
 5. 接下来，Shell 会调用 `write` 函数（或 `printf` 函数）将字符输出到屏幕。所以，在 `/kernel/chr_drv/tty_io.c` 文件中 `tty_write` 函数的第 407 行添加一个断点，并删除其它所有的断点。
 6. 按 F5 继续调试，会在刚刚添加的断点处中断。将鼠标放到 `tty_write` 函数的第一个参数上，显示其值为 0，在第 409 行根据此参数的值获取到了 `tty0`。
 7. 选择“调试”菜单“窗口”中的“调用堆栈”，打开调用堆栈窗口，双击其中的 `sys_write` 函数调用帧，可以切换到 `sys_write` 函数中。此时，将鼠标移动到 `sys_write` 函数的第一个参数 `fd` 上，显示其值为 2，说明 Shell 程序确实调用了 `write` 函数，并且传入的文件句柄为 2，也就是 `stderr`。
 8. 按 F10 单步调试，直到在第 448 行停止。其中，第 419 行是从用户缓冲区中读取一个字符到变量 `c` 中；第 444 行将变量 `c` 中的字符放入 `tty0` 的写队列 `write_q` 中。使用“快速监视”对话框查看表达式“`*tty`”的值，可以查看队列 `write_q` 中的内容，可以看到队尾的字符 `a`。
 9. 由于 `tty0` 的 `write` 函数指针指向文件 `kernel/chr_drv/console.c` 中的控制台写函数 `con_write`。所以，按 F11 单步调试，会进入 `con_write` 函数并中断。
 10. 按 F10 单步调试，直到在第 616 行停止。第 615 行是从 `tty0` 的写队列 `write_q` 中取一个字符放入变量 `c` 中，将鼠标移动到源代码中变量 `c` 的名称上，会显示变量 `c` 的值为 `0x61` 即字符“a”。
 11. 按 F10 继续调试，直到第 627 行停止。第 627 行的汇编代码是将字符直接写到显示内存中的指定位置，使字符显示到屏幕上。此时查看 Bochs 虚拟机的 Display 窗口，可以看到字符“a”还没有显示出来。再次按 F10 单步调试一次，这段汇编代码就会执行，再次查看 Bochs 虚拟机的 Display 窗口，可以看到字符“a”已经显示出来了。
 12. 删除所有断点后，结束此次调试。

刚刚的调试过程是键盘按键按下的处理过程。读者可以自己尝试调试键盘抬起（产生断开码）的处理过程。可以在 `kernel/chr_drv/keyboard.s` 文件的第 89 行添加一个条件断点，条件设置为“`$eax==0x9e`”（其中 `0x9e` 是按键 A 的断开码）。添加条件断点的具体方法可以参考实验四的第 3.4 节。

3.3 字符显示的控制

通过上面的调试过程，读者应该对键盘中断处理过程有了初步了解。下面的实验是修改 Linux 0.11 的键盘设备和显示设备处理代码，对键盘输入和字符显示进行非常规的控制：在 Linux 0.11 刚启动时，一切如常；当按一次 F12 后，向终端输出的所有字母和数字都会被替换为“*”字符；再按一次 F12 后，又恢复正常；第三次按 F12 后，再进行替换，依此类推。

基本思路是在 Linux 0.11 内核中添加一个全局变量作为进行字符替换操作的标志，初始

值为0，表示不进行字符替换。当用户按下F12键后，将此标志置为1，然后在将字符输出到显示设备的con_write函数中进行字符替换。当用户再次按下F12键后，将标志恢复为0。

请读者按照下面的步骤进行实验：

1. 将 kernel/chr_drv/keyboard.s 文件中的第 274 行注释掉。这样，当按下键盘上的 F12 时，系统就不会调用显示进程列表的函数了。
2. 在 include/asm/system.h 文件的第三行前添加一个全局变量的声明：

```
extern int judge;
```

此变量是用作判断标志。初始值为0，当F12键被按下时judge被设置为1，F12键再次被按下时，judge又被设置为0。如此循环。
3. 在 kernel/chr_drv/tty_io.c 文件的第61行添加judge的定义：

```
int judge = 0;
```

并修改 copy_to_cooked 函数，在第196行 GETCH(tty->read_q, c); 语句的后面添加如下代码：

```
if(c=='L')
{
    if(judge) judge=0;
    else judge=1;
    break;
}
```

这样，在取到字符后，如果字符为“L”，表示 F12被按下（按键F12的接通码会转换为ASCII码76，即字母“L”）。
4. 修改 kernel/chr_drv/console.c 文件中的 con_write 函数，在第615行后面添加如下代码，如果字符是英文字母或者阿拉伯数字，就将之替换为“*”号：

```
if(judge)
{
    if((c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' && c<='9'))
        c=42;
}
```
5. 按F7生成项目，然后按F5启动调试。
6. 执行ls命令，可以正常看到当前目录下的所有文件。
7. 按F12，此时再执行ls命令，可以看到，所有的英文字母和阿拉伯数字全都变成了“*”。
8. 再按F12，再执行ls命令，可以看到，一切又回归到正常。
9. 修改步骤3中的源代码，将字母“L”换成其他字母（F1-F12分别对应‘A’-‘L’），重新调试，试试效果。
10. 修改步骤4中的语句“c=42;”，将 42换成其他ASCII码值，重新调试，再试试效果。

本实验设计的只是将英文字母与阿拉伯数字转换成了其他字符，读者可以通过修改步骤4中的源代码将任意字符（例如标点符号）替换为读者想要的字符。

3.4 实现贪吃蛇游戏

通过上面的实验，读者应该已经掌握了键盘中断响应的处理过程。接下来，请读者按照下面的步骤来实现一个具有一定可玩性的贪吃蛇游戏：

1. 在文件 include/asm/system.h 的第三行添加一个全局变量的声明：

```
extern int fflag;
```

此变量是用作判断标志。初始值为0，表示贪吃蛇游戏未启动，当F2键被按下时设

置为1，表示贪吃蛇游戏启动。

- 2、在文件kernel/chr_drv/tty_io.c中的第62行添加全局变量的定义：

```
int fflag = 0;
```

并在第202行的后面添加如下代码。其作用是当F2键被按下时，将标志设置为1，启动贪吃蛇游戏。F5、F6、F7、F8键分别代表上下左右四个方向，分别将标志设置为2、3、4、5。

```
if(c=='B') //F2启动游戏
```

```
{
    fflag = 1;
    break;
}
```

```
if(c == 'E') //F5上
```

```
{
    fflag = 2;
    break;
}
```

```
if(c == 'F') //F6下
```

```
{
    fflag = 3;
    break;
}
```

```
if(c == 'G') //F7左
```

```
{
    fflag = 4;
    break;
}
```

```
if(c == 'H') //F8右
```

```
{
    fflag = 5;
    break;
}
```

- 3、在kernel/chr_drv/console.c文件的第598行后面(con_write函数的前面)添加如下代码。函数snake_move将蛇头字符设置在显示内存中由 pos 指定的位置，从而显示在屏幕上。snake_stop函数中使用全局变量jiffies完成一个计时循环(jiffies可参考实验六中的说明)，从而让贪吃蛇停止一定的时间。

//将蛇头字符设置在显示内存中由 pos 指定的位置，从而显示在屏幕上

```
void snake_move()
```

```
{
    __asm__ ("movb _attr,%ah\n\t"
            "movw %%ax,%l\n\t"
            :: "a" ('+'), "m" (*(short *)pos)
            );
}
```

```
//让贪吃蛇停留一定的时间
void snake_stop()
{
    int i;
    for(i = jiffies + 50; jiffies <= i;)
        ;
}
```

- 4、在kernel/chr_drv/console.c文件的第637行后面(con_write函数中第一个switch语句的前面)添加如下代码，根据标志位来启动游戏，并且让贪吃蛇向指定的方向移动。

```
switch(fflag)
{
    case 0:break;
    case 1: //初始化
        csi_J(2); //清屏
        x = 0; y = 0;
        gotoxy(x,y); //将当前光标设置为屏幕左上角的坐标
        set_cursor(); //将光标移动到屏幕左上角
    case 5: //向右移动
        while(fflag == 5 || fflag == 1)
        {
            delete_line(); //删除光标所在的行
            x < video_num_columns-1 ? x++ : x; //光标坐标向右移动一位
            gotoxy(x,y); //更新光标位置
            set_cursor(); //设置显示器光标的位置
            snake_move(); //蛇向右移动
            snake_stop(); //停留
        }
        break;
    case 3://向下移动
        while(fflag == 3)
        {
            delete_line(); //删除光标所在的行
            y < video_num_lines-1 ? y++ : y; //光标坐标向下移动一位
            gotoxy(x,y); //更新光标位置
            set_cursor(); //设置显示器光标的位置
            snake_move(); //蛇向下移动
            snake_stop(); //停留
        }
        break;
}
```

- 5、按F7进行编译，按F5启动调试。
- 6、按F2启动贪吃蛇游戏。游戏启动后，贪吃蛇从屏幕的左上角出现并自动向右移动。此时可以按F6将贪吃蛇的移动方向改为向下。再按F8即可将贪吃蛇的移动方向改为

向右。

目前只实现了将贪吃蛇向右移动和向下移动的基础功能,请读者在充分理解之前添加的代码的基础上,为贪吃蛇游戏添加下面的功能,使其具有一定的可玩性:

- a) 添加贪吃蛇向上和向左移动的功能。
- b) 当蛇头“+”移动到屏幕的边缘时,就会在与之相反的边缘出现,继续同方向移动。
- c) 在屏幕上的某些位置出现“#”字符,当蛇头“+”与“#”相遇后,“#”消失,并且在另外一个位置再出现一个“#”。同时,蛇的尾部就多出一个“*”,作为蛇身,吃的“#”越多,蛇身就越长。
- d) 蛇身越长,贪吃蛇移动的速度越快。
- e) 当蛇头“+”撞到蛇身“*”后,结束游戏。
- f) 同时出现2个贪吃蛇,实现双人对战,甚至多人对战。

实验十一 proc 文件系统的实现

实验性质：验证、设计

建议学时：2 学时

实验难度：★★★☆☆

一、实验目的

- 掌握proc文件系统的实现原理。
- 掌握文件、目录、索引节点等概念。

二、预备知识

最新的Linux内核通过文件系统接口实现了proc文件系统，它是一个虚拟文件系统，在Linux启动时就被挂接（mount）到了/proc目录上。proc通过虚拟文件和虚拟目录的方式提供访问系统参数的机会，所以有人称它为“了解系统信息的一个窗口”。这些虚拟的文件和目录并没有真实的存在于磁盘上，而是在内存中形成了一种对Linux内核数据的直观表示，并且随着操作系统的运行自动建立、删除和更新。虽然是虚拟的，但它们都可以通过标准的文件系统调用来访问（包括read、write函数等）。

Linux 0.11还没有实现虚拟文件系统，也就是还没有提供增加新文件系统类型的接口。所以本实验只能在现有文件系统的基础上，通过打补丁的方式模拟一个proc文件系统。Linux 0.11使用的是MINIX 1.0文件系统，这是一种典型的基于i节点(inode)的文件系统，《Linux内核完全注释》一书的第12章对它有详细描述。MINIX 1.0文件系统系统中的每个文件都要对应至少一个inode，而inode中记录着文件的各种属性，包括文件类型等。文件类型有普通文件、目录、字符设备文件和块设备文件等。在Linux 0.11的内核中，每种类型的文件都有不同的处理函数与之对应。所以，可以在MINIX 1.0文件系统中增加一种新的文件类型——proc文件，并在相应的处理函数内实现proc文件系统的功能。

三、实验内容

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 新建一个 Linux011 Kernel 项目。

3.2 实现 proc 文件系统

下面的内容会引导读者实现一个简单的 proc 文件系统，并在其中添加一个用于显示Linux 0.11 操作系统进程信息的 psinfo 节点。每当进程信息发生变化时，Linux 0.11 操作系统的内核负责向 psinfo 节点中写入相关的数据，然后 Linux 0.11 的应用程序就可以从 psinfo 节点中读取数据了。读取数据的方式与从普通文件中读取数据的方式完全相同。

请读者按照下面的步骤进行实验：

1. 在 MINIX 1.0 文件系统中为 proc 文件增加一个新文件类型。在 include/sys/stat.h 文件的第 25 行后面添加新文件类型的宏定义，其代码如下：

```
#define S_IFPROC 0050000
```

新文件类型宏定义的值应该在 0010000 到 0100000 之间，但后四位八进制数必须是 0，而且不能和已有的任意一个 S_IFXXX 相同

2. 为新文件类型添加相应的测试宏。在第 37 行后面添加测试宏，代码如下：

```
#define S_ISPROC(m) ((m) & S_IFMT) == S_IFPROC)
```

3. psinfo 结点要通过 mknod 系统调用建立，所以要让它支持新的文件类型。在 fs/namei.c 文件中只需修改 mknod 函数中的一行代码即可，即将第 647 行代码修改为如下：

```
if(S_ISBLK(mode) || S_ISCHR(mode) || S_ISPROC(mode))
```

proc 文件系统的初始化工作应该在根文件系统被挂载之后开始，包括两个步骤：建立 /proc 目录和建立 /proc 目录下的各个结点（本实验只建立 /proc/psinfo 结点）。因为在根文件系统加载后，初始化过程已经进入用户态，就不能调用 sys_mkdir 和 sys_mknod 函数建立目录和结点了，而必须调用 mkdir 和 mknod 系统调用函数来建立目录和结点。所以，必须在初始化代码所在文件中实现 mkdir 和 mknod 两个系统调用函数的用户态接口。请读者按下面的步骤继续修改 Linux 0.11 内核的源代码：

1. 在 init/main.c 文件的第 13 行包含头文件 stat.h：

```
#include<sys/stat.h>
```

在第 42 行后面添加如下代码，定义 mkdir 和 mknod 两个系统调用函数：

```
_syscall2(int, mkdir, const char*, name, mode_t, mode)
```

```
_syscall3(int, mknod, const char*, filename, mode_t, mode, dev_t, dev)
```

在第 234 行复制句柄语句的后面添加如下代码，调用 mkdir 函数创建 /proc 目录，调用 mknod 函数创建 psinfo 结点：

```
mkdir("/proc", 0755);
```

```
mknod("/proc/psinfo", S_IFPROC|0444, 0);
```

由于 proc 文件系统对用户态程序来说是只读的，所以将 mkdir 函数的第二个参数 mode 的值设为“0755”（rwxr-xr-x），表示只允许 root 用户改写此目录，其它用户只能进入和读取此目录。将 mknod 函数的第二个参数 mode 的值设为“S_IFPROC|0444”，表示这是一个 proc 文件，权限为 0444（r--r--r--），对所有用户只读。

mknod 函数的第三个参数 dev 用来说明结点所代表的设备编号。对于 proc 文件系统来说，此编号可以完全自定义。proc 文件的处理函数可以通过这个编号决定文件包含的是什么信息。例如，可以把 0 对应 psinfo，1 对应 meminfo，2 对应 cpuinfo。

2. 找到 fs/read_write.c 文件中的 sys_read 函数，此函数中有一系列的 if 判断语句，再添加一个 if 语句，代码如下：

```
if(S_ISPROC(inode->i_mode))
```

```
return psread(inode->i_zone[0], buf, count, &file->f_pos);
```

这样，当文件类型为 proc 文件时，就会调用 psread 函数进行了。传递给 psread 函数的参数包括：

- inode->i_zone[0]，这就是之前调用 mknod 函数时指定的 dev —— 设备编号。
- buf，用户空间缓冲区，就是应用程序在调用 read 函数时传入的第二个参数，用于存放从文件中读取到的数据。
- count，就是应用程序在调用 read 函数时传入的第三个参数，用于说明 buf 指向的缓冲区的大小。
- &file->f_pos，f_pos 是上一次读文件结束时“文件位置指针”的位置。这里必须传递指针，因为 psread 函数还需要根据读取到的数据量修改 f_pos 的值。

3. 由于这里调用了 psread 函数，所以需要在第 32 行添加如下代码声明此函数：


```
extern int psread (int dev, char * buf, int count, off_t * f_pos);
```

psread 函数的源代码在“学生包”本实验对应文件夹下的 proc.c 文件中。请读者按照下面的步骤将 proc.c 文件中的源代码添加到 Linux 0.11 的内核中：

1. 在Linux Lab的“项目管理器”窗口中，右键点击“fs”文件夹节点，选择菜单“添加”中的“添加新文件”，打开“添加新文件”对话框。
2. 在“添加新文件”对话框中选中“C源文件(.c)”模板，输入文件名称“procfs.c”后点击“添加”按钮，添加一个新源文件procfs.c。
3. 将“学生包”本实验对应文件夹下的proc.c文件拖动到Linux Lab中释放，即可打开此文件。将其中的代码全部复制到刚刚新建的源文件procfs.c中。
4. 按F7键生成项目，确保没有语法错误。
5. 按F5启动调试，待Linux 0.11启动以后，输入命令：`ls -l /proc`可以查看/proc目录的属性信息，如下图所示：

```
total 0
-r--r--r--  1 root    root          0 ??? ?  ??? psinfo
```

图 11-1 proc 文件夹内的文件属性信息

输入命令：`cat /proc/psinfo` 即可得到当前所有进程的信息，如下图所示：

```
pid      state  father  counter start_time
0         1       -1       0         0
1         1       0        28        1
4         1       1         1        71
3         1       1        24        65
6         0       4        12       785
```

图 11-2 打印输出当前所有进程的信息

3.3 调试 proc 文件系统的工作过程

请读者按照下面的步骤调试 proc 文件系统的工作过程，进而理解相关的源代码：

1. 在 fs/procfs.c 文件中 psread 函数的第一个 if 语句处(第 70 行)添加一个断点。
2. 按 F5 启动调试，待 Linux 0.11 启动后，输入命令“`cat /proc/psinfo`”后按回车，会在刚刚添加的断点处中断。此 if 语句的作用是：当第一次调用 psread 函数时，文件位置指针 f_pos 的值为 0，就会进入这个 if 语句将进程信息放入缓冲区 psbuffer 中。但是，如果需要再次进入 psread 函数时，由于文件位置指针 f_pos 的值不为 0，就不再进入这个 if 语句，从而可以直接跳转到后面的 for 循环读取数据了。
3. 按 F10 单步调试到第 81 行，这个过程中调用了五次 addTitle 函数将标题添加到了 psbuffer 缓冲区中。接下来第 85 行开始的 for 循环，会遍历进程控制块数组，将有效的进程信息写入 psbuffer 缓冲区中。
4. 在 psread 函数的最后一个 for 循环语句处(第 102 行)添加一个断点。按 F5 继续执行，程序就会命中此断点。此 for 循环的目的是将 psbbuffer 缓冲区内的数据逐字节的读取到用户空间中的 buf 缓冲区中。
5. 在 psread 函数最后的 return 语句处添加一个断点，按 F5 继续调试，程序会在此处中断。将鼠标移动到返回值变量 i 上可以查看此变量的值，再将鼠标移动到 psread 函数的第三个参数 count 上可以查看此变量的值，通过比较这两个值可以发现，psread 函数实际读取的字节数小于 buf 缓冲区的大小 count，也就是说 psread 函数执行一次就可以读取到所有数据了。

请读者考虑一下 cat 命令的实现过程（可以参考下面的源代码），虽然调用一次 read 函数就已经读取到了所有数据，但是由于 cat 实现代码中的 while 循环第一次读到的字节数 nread 不为 0，所以会再次执行 read 函数来读取文件，也就会再次进入 psread 函数。

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char* argv[])
{
    char buf[513] = {'\0'};
    int nread;
    int fd = open(argv[1], O_RDONLY, 0);
    while(nread = read(fd, buf, 512))
    {
        buf[nread] = '\0';
        puts(buf);
    }
    return 0;
}
```

请读者按照下面的步骤继续调试：

1. 按 F5 继续调试，程序会在第 70 行的断点处再次中断。此时查看 Bochs 虚拟机的显示窗口，已经打印输出 psinfo 节点的内容了。
2. 继续按 F10 单步调试，可以发现程序并没有进入第一个 if 语句，而且没有读取任何数据就从第 111 行的 break 语句跳出了 for 循环，最后返回了 0。
3. 按 F5 继续运行，cat 命令就结束了，也就不会再命中任何断点了。

3.4 简化 fs/procfs.c 文件中的源代码

文件 fs/procfs.c 中 psread 函数的源代码调用了 itoa 和 addTitle 函数将内容写入 psbuffer 缓冲区，造成源代码比较复杂。读者可以使用 sprintf 函数替换掉 itoa 和 addTitle 函数，源代码就会简单很多。但是 Linux 0.11 没有实现 sprintf 函数，读者可以参考 init/main.c 文件中第 200 行的 printf 函数，在 fs/procfs.c 文件中实现一个 sprintf 函数。

四、思考与练习

1. 请读者模仿 psinfo 节点的实现方法，实现一个保存物理内存信息的 meminfo 节点，该节点保存的数据可以参考实验八第 3.2 节打印输出的物理内存的信息。
2. 实现一个可以读取 jiffies 时钟滴答值的 tickinfo 节点。时钟滴答 jiffies 的说明可以参考实验六第 3.3.3 节的内容。考虑到 jiffies 的值改变的太快了（10ms 变一次），能够直接将 Linux 0.11 中全局变量 jiffies 的值转换为字符串并放入一个缓冲区中供用户读取吗？在将 jiffies 的值转换为字符串的过程中，其值是否有可能发生改变呢？如何解决这个问题。
3. 在读取节点 psinfo 的过程中，函数 psread 会将所有进程的信息转换为字符串并放入 psbuffer 缓冲区中，但是在循环遍历所有 64 个进程的过程中，是否会出现进程的信息发生变化的情况呢？如果会发生变化，读者是否可以参考前一个练习解决 jiffies 的值变化太快的方法来解决此问题。

实验十二 MINIX 1.0 文件系统的实现

实验性质：验证、设计

建议学时：2 学时

实验难度：★★★★☆

一、实验目的

- 通过查看MINIX 1.0文件系统的硬盘信息，理解MINIX 1.0的硬盘管理方式。
- 学习MINIX 1.0文件系统的实现方法。
- 改进MINIX 1.0文件系统的实现方法，加深对MINIX 1.0文件系统的理解。

二、预备知识

Linux 0.11操作系统启动时需要加载一个根目录，此根目录使用的是MINIX 1.0文件系统，其保存在硬盘的第一个分区中。Linux 0.11操作系统将硬盘上的两个连续的物理扇区（大小为512字节）做为一个物理盘块（大小为1024字节），而且物理盘块是从1开始计数的。硬盘上的第一个物理盘块是主引导记录块（MBR），读者已经通过实验二了解到，Linux 0.11并未使用硬盘上的主引导记录块进行引导，而是使用软盘A上的引导扇区进行引导的。在硬盘的主引导记录块中除了没有用到的引导程序外，在其包含的第一个物理扇区的尾部（引导标识0x55aa的前面）是一个64字节的分区表（典型的IBM Partition Table），其中每个分区表项占用16字节，共有4个分区表项。每个分区表项都定义了一个分区的起始物理盘块号和分区大小（占用的物理盘块数量）等信息。硬盘上的物理盘块可以按照图12-1所示进行划分。

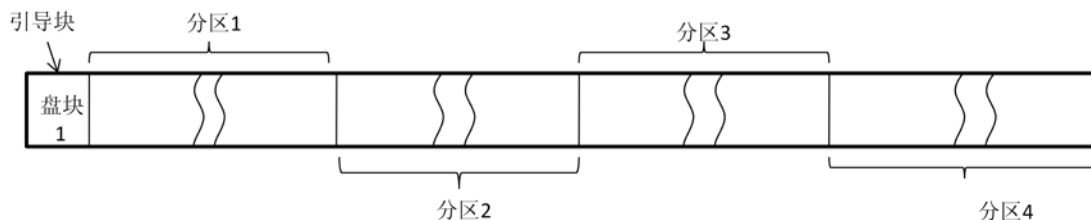


图 12-1 硬盘上物理盘块的划分

Linux 0.11使用的硬盘只包含了两个分区，在第一个分区中提供了Linux 0.11的根目录，并使用MINIX 1.0文件系统进行管理，第二个分区没有用到，内容为空。需要强调的是，在MINIX 1.0文件系统管理的硬盘分区中就不再使用物理盘块作为划分的单位了，而是将一个物理盘块作为一个逻辑块来使用，并且逻辑块号是从0开始计数的。

接下来，重点了解一下MINIX 1.0文件系统是如何管理硬盘上的第一个分区的。第一个分区包含了引导块（占用逻辑块0）、超级块（占用逻辑块1）、i节点位图（占用逻辑块2-4）、逻辑块位图（占用逻辑块5-12），以及i节点和数据区，如图12-2所示。



图 12-2 MINIX 1.0 文件系统所管理的分区 1 的布局

分区 1 开始处的引导块同样没有被使用。其后的超级块用于存放 MINIX 1.0 文件系统的结构信息，主要用于说明各部分的起始逻辑块号和大小（包含的逻辑块数量）。超级块中各个字段的含义在表 12-3 中进行了说明。

字段名称	数据类型	说明
s_inodes	short	分区中的 i 节点总数。
s_nzones	short	分区包含的逻辑块总数。
s_imap_blocks	short	i 节点位图占用的逻辑块数。
s_zmap_blocks	short	逻辑块位图占用的逻辑块数。
s_firstdatazone	short	数据区占用的第一个逻辑块的块号。
s_log_zine_size	short	log2(逻辑块包含的物理块数量)。MINIX 1.0 文件系统的逻辑块包含一个物理块，所以其值为 0。
s_max_size	long	文件最大长度，以字节为单位。
s_magic	short	文件系统魔数，用以指明文件的类型。MINIX 1.0 文件系统的魔数是 0x137f。

表 12-3 MINIX 1.0 的超级块的结构

在超级块的后面是占用了 3 个逻辑块的 i 节点位图，其中的每一位用于说明对应的 i 节点是否被使用。位的值为 0 时，表示其对应的 i 节点未被使用；位的值为 1 时，表示其对应的 i 节点已经被使用。

在 i 节点位图的后面是占用了 8 个逻辑块的逻辑块位图，其中的每一位用于说明数据区中对应的逻辑块是否被使用。除第 1 位(位 0)未被使用外，逻辑块位图中每个位依次代表数据区中的一个逻辑块。因此，逻辑块位图的第 2 位(位 1)代表数据区中第一个逻辑块，第 3 位(位 2)代表数据区中的第二个逻辑块，依此类推。当数据区中的一个逻辑块被占用时，逻辑块位图中的对应位被置为 1，否则被置为 0。

在逻辑块位图的后面是 i 节点部分，其中存放着 MINIX 1.0 文件系统中文件或目录的索引节点(简称 i 节点)。注意，i 节点是从 1 开始计数的。每个文件或目录都有一个 i 节点，每个 i 节点结构中存放着对应文件或目录的相关信息，如文件宿主的 id(uid)、文件所属组 id(gid)、文件长度、访问修改时间以及文件数据在数据区中的位置等。i 节点共包含 32 个字节，其结构如表 12-4 所示。

字段名称	数据类型	说明
i_mode	short	文件的类型和属性(rwx 位)
i_uid	short	文件宿主的用户 id
i_size	long	文件长度(以字节为单位)
i_mtime	long	文件的修改时间(从 1970 年 1 月 1 日 0 时起，以秒为单位)
i_gid	char	文件宿主的 id
i_nlinks	char	链接数(有多少个文件目录项指向该 i 节点)
i_zone[9]	short	文件所占用的数据区中的逻辑块号数组。其中 zone[0]~zone[6]是直接块号；zone[7]是一次间接块号；zone[8]是二次(双重)间接块号。

表 12-4 MINIX 1.0 文件系统的 i 节点的结构

i_mode 字段用来保存文件的类型和访问权限属性。其位 15~12 用于保存文件类型，位 11~9 保存执行文件时设置的信息，位 8~0 表示文件的访问权限。如图 12-5 所示。i_zone[] 数组用于存放 i 节点在数据区中对应的逻辑块号。i_zone[0]到 i_zone[6]用于存放文件开始的 7 个块号，称为直接块。例如，若文件长度小于等于 7KB，则根据其 i 节点的 i_zone[0]

涉及到硬盘设备的物理特性和驱动程序代码。为了简化实验内容，这里采用模拟的方式来访问 MINIX 1.0 文件系统，即使用 C 语言编写一个 Windows 控制台程序，直接访问 Linux Lab 提供的硬盘镜像文件中的 MINIX 1.0 文件系统。感兴趣的读者可以在完成本实验后，继续阅读 Linux 0.11 的源代码，学习 Linux 0.11 操作系统是如何访问硬盘设备，以及硬盘分区中的文件系统的。

3.1 准备实验

1. 启动 Engintime Linux Lab。
2. 使用“控制台应用程序(C)”模板，新建一个 Windows 控制台应用程序项目。

3.2 打印输出 MINIX 1.0 文件系统的目录树

在“学生包”本实验对应文件夹中，提供了一个源代码文件 minix.c，其中实现了打印输出 MINIX 1.0 文件系统的目录树的功能。使用 Linux Lab 打开此文件(将文件拖放到 Linux Lab 中释放即可打开)，仔细阅读其中的源代码，并着重理解下面的内容：

- 在源代码文件的开始位置定义了分区表项、超级块、i 节点和目录项的结构体，读者需要理解其中每个字段的意义。
- get_physical_block 函数的作用是将一个物理块的内容读取到缓冲区中。get_partition_logical_block 函数的作用是将第一个分区中的一个逻辑块的内容读取到缓冲区中。需要注意的是，物理块号是从 1 开始计数的，而逻辑块号是从 0 开始计数的。这两个函数实现了对硬盘镜像文件物理层和逻辑层的分层访问，从而可以模拟出访问硬盘的过程。
- load_inode_bitmap 函数将硬盘镜像文件中的整个 i 节点位图都读入到了内存中。is_inode_valid 函数根据 i 节点位图中的内容判断一个 i 节点是否有效。需要注意的是，i 节点是从 1 开始计数的。get_inode 函数根据 i 结点的 id 读取相对应的 i 结点。
- print_inode 函数递归打印目录树。首先读取 i 节点位图的第一位，并找到根节点。然后判断 i 节点是否为目录，若是目录，则打印目录名(名称前加 Tab 键是为了有树的层次感)，然后递归打印其子目录和子文件；如果是常规文件则直接打印文件名。

请读者按照下面的步骤查看 MINIX 1.0 文件系统的目录树：

1. 将 minix.c 文件中的源代码复制并替换到新建的控制台应用程序项目的源代码文件中。
2. 确保 main 函数中需要打开的 harddisk.img 文件存在。如果不存在，需创建一个 Linux 0.11 的内核项目，然后将其项目文件夹中的 harddisk.img 文件复制到 path 字符串变量指定的磁盘位置。
3. 按 F7 生成项目，确保没有语法错误。
4. 打开本项目的文件夹，将 Debug 文件夹下的 console.exe 文件复制到 C:\minix 目录下。
5. 启动 Windows 控制台，进入 C:\minix 目录，然后执行命令“console.exe > a.txt”。此命令会将应用程序打印输出的目录树重定向到文本文件 a.txt 中。
6. 打开 a.txt 文件查看 MINIX 1.0 文件系统的目录树。

3.3 实现更多功能

1. 参考 load_inode_bitmap 函数写出一个可以将硬盘镜像文件中的整个逻辑块位图都加载到内存中的函数。

2. 打印输出类似于 Linux 0.11 内核命令“df”的输出内容。
3. 将“/usr/root”文件夹下的“hello.c”文件的内容打印输出。
4. 将“/usr/src/linux-0.11.org”文件夹下的“memory.c”的内容打印输出。注意，此文件大于 7KB，所以需要使用二次间接块才能访问所有的数据。
5. 删除“/usr/root/hello.c”文件。
6. 删除“/usr/root/shoe”文件夹。
7. 新建“/usr/root/dir”文件夹。
8. 新建“/usr/root/file.txt”文件，并设置初始大小为 10KB。

参考文献

- [1] 刘刚, 赵鹏翀著。操作系统实验教程。北京: 清华大学出版社, 2013
- [2] 汤子瀛, 哲凤屏, 汤小丹著。计算机操作系统。西安: 西安电子科技大学出版社, 1996
- [3] [美] Andrew S. Tanenbaum, Albert S. Woodhull 著。操作系统: 设计与实现 (第二版) 上册。王鹏, 尤晋元, 朱鹏, 敖青云译。北京: 电子工业出版社, 1998
- [4] 赵炯著。Linux 内核完全剖析。北京: 机械工业出版社, 2006
- [5] [英] Peter Abel 著。IBM PC 汇编语言程序设计 (第五版)。沈美明, 温冬婵译。北京: 人民邮电出版社, 2002
- [6] 刘星等著。计算机接口技术。北京: 机械工业出版社, 2003
- [7] 唐朔飞著。计算机组成原理。北京: 高等教育出版社, 2000
- [8] [希腊] Diomidis Spinellis 著。代码阅读方法与实践。赵学良译。北京: 清华大学出版社, 2004
- [9] [美] 科学、工程和公共政策委员会著。怎样当一名科学家。何传启译。北京: 科学出版社, 1996
- [10] <https://cms.hit.edu.cn/course/view.php?id=44> 哈尔滨工业大学操作系统实验课
- [11] Intel Co. INTEL 80386 Programmes's Refercence Manual 1986, INTEL CORPORATION, 1987
- [12] Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume. 3: System Programming Guide. <http://www.intel.com/>, 2005
- [13] IEEE-CS, ACM. Computing Curricula 2001 Computer Science. 2001
- [14] The NASM Development Team. NASM — The Netwide Assembler. Version 2.04 2008
- [15] Bochs simulation system. <http://bochs.sourceforge.net/>